8. Chart Parsing

8.1 Introduction

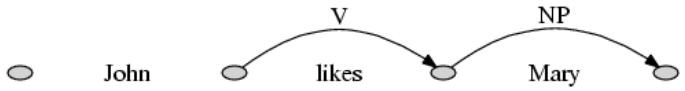
The simple parsers discussed in the parsing tutorial have significant limitations. The bottom-up shift-reduce parser can only find one parse, and it often fails to find a parse even if one exits. The top-down recursive-descent parser can be very inefficient, since it often builds and discards the same sub-structure many times over; and if the grammar contains left-recursive rules, it can enter into an infinite loop.

These completeness and efficiency problems can be addressed by employing a technique called *dynamic programming*, which stores intermediate results, and re-uses them when appropriate.

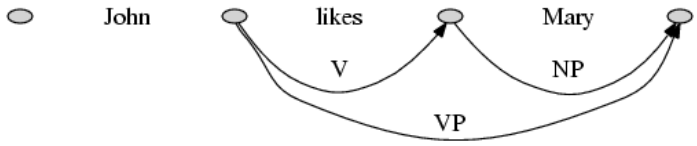
In general, a parser hypothesizes constituents based on the grammar and its current knowledge about the tokens it has seen and the constituents it has already found. Any constituent that is consistent with the current knowledge can be hypothesized; but many of these hypothesized constituents may not be used in complete parses.

A *chart parser* uses a structure called a *chart* to record the hypothesized constituents in a sentence. One way to envision this chart is as a graph whose nodes are the word boundaries in a sentence. For example, an empty chart for the sentence "John likes Mary" can be drawn as follows:

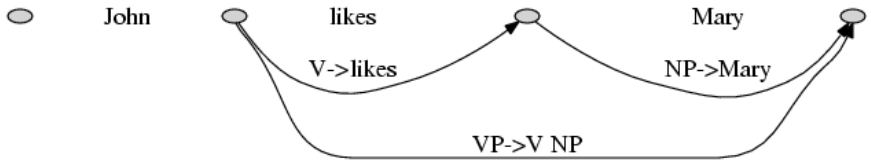
Each hypothesized constituent is drawn as an *edge* in this graph. For example, the following chart hypothesizes that "likes" is a V and "Mary" is an NP:



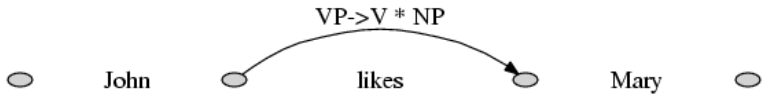
And the following chart also hypothesizes that "likes Mary" is a VP:



In addition to recording a constituent's type, it is also useful to record the types of its children. In other words, we can associate a single CFG production with an edge, rather than just its nonterminal type:



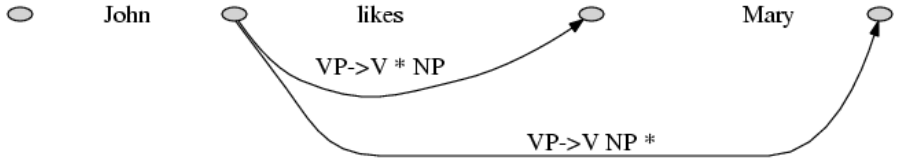
All of the edges that we've seen so far represent complete constituents. However, it can also be helpful to hypothesize *incomplete* constituents. For example, we might want to record the hypothesis that "the V constituent *likes* forms the beginning of a VP." We can record hypotheses of this form by adding a *dot* to the edge's right hand side. The children to the left of the dot specify what children the constituent starts with; and the children to the right of the dot specify what children still need to be found in order to form a complete constituent. For example, the edge in the following chart records the hypothesis that "a VP starts with the V *likes*, but still needs an NP to become complete":



These *dotted edges* are used to record all of the hypotheses that a chart parser makes about constituents in a sentence. Formally, we can define a dotted edge as follows:

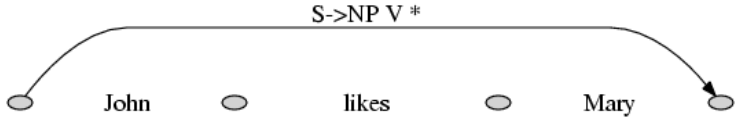
> A dotted edge [A → c1 ... cd; • cd+1 ... cn]@[i:j] records the hypothesis that a constituent of type A starts with children c1...cd covering words wi...wj, but still needs children cd+1...cn to be complete (where both c1...cd and cd+1...cn may be empty.)

If d=n (i.e., if cd+1...cn is empty) then 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 the following chart, [VP → V NP •]@[1:3] is a complete edge, and [VP → V • NP]@[1:2] is an incomplete edge.



If n=0 (i.e., if c1...cn is empty), then the edge is called a *self-loop edge*. In the following chart, [VP → • V NP]@[1:1] is a self-loop edge.

If a complete edge spans the entire sentence, and has the grammars' 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 the following chart, [S → NP VP •]@[0:3] is a parse edge.



8.2 Chart Parsing

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:

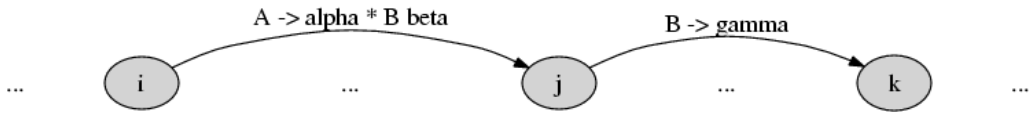
1. The *sentence* can license an edge. In particular, each word wi in the sentence licenses the complete edge [wi → •]@[i:i+1].
2. The *grammar* can license an edge. In particular, each grammar production A → α licenses the self-loop edge [A → • α]@[i:i] for every i, 0≤i<n.
3. The *current chart contents* 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:

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*.

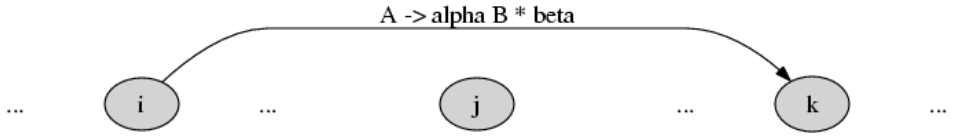
8.2.1 The Fundamental Rule

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 complete edge immediately following it whose left hand side is B. Formally, it states that if the chart contains the edges:



1. [A → α • B β]@[i:j]
2. [B → γ •]@[j:k]

Then the parser should add the edge:

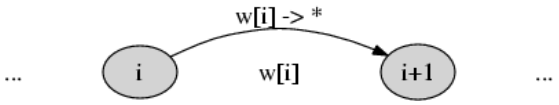


3. [A → α B • β]@[i:k]

8.2.2 Bottom Up Parsing

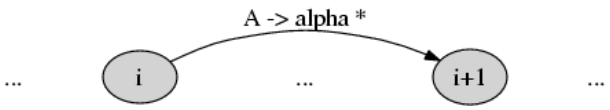
To create a bottom-up parser, we need to add two rules: the *Bottom-Up Initialization Rule*; and the *Bottom-Up Predict Rule*.

The Bottom-Up Initialization Rule says to add all edges licensed by the sentence. In particular, it states that for every word w_i, the parser should add the edge:



1. $[w_i; \rightarrow \bullet]@[i:i+1]$

The Bottom-Up Predict Rule says that if the chart contains a complete edge, then the parser add a self-loop edge at the complete edge's left boundary for each grammar production whose right-hand side begins with the completed edge's left-hand side. In other words, it states that if the chart contains the complete edge:



1. $[A \rightarrow \alpha \bullet]@[i:j]$

And the grammar contains the production:

2. $B \rightarrow A\ \beta$

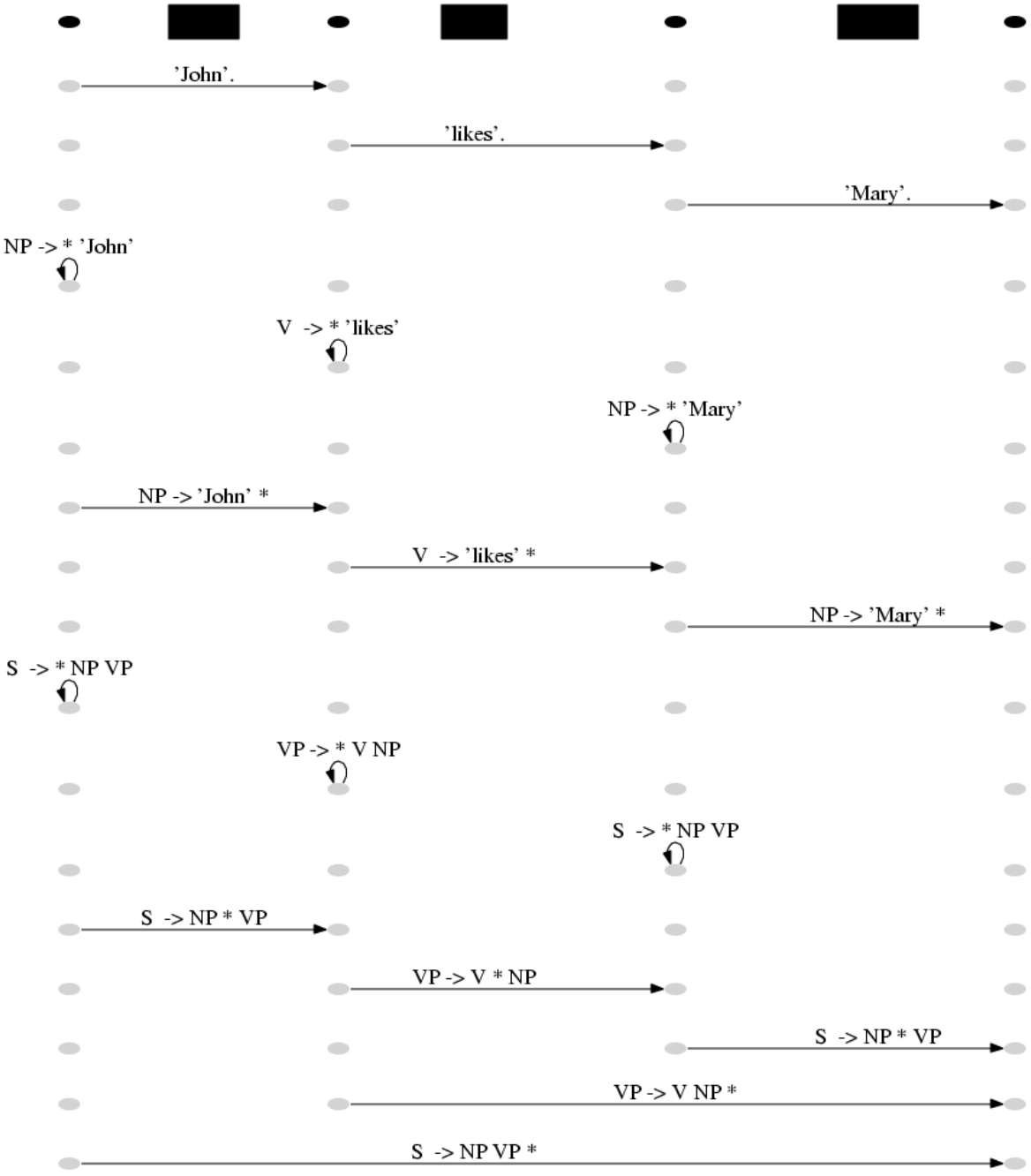
Then the parser should add the self-loop edge:

3. $[B \rightarrow \bullet\ \beta]@[i:i]$

Using these three rules, we can parse a sentence as follows:

1. Create an empty chart spanning the sentence.
2. Apply the Bottom-Up Initialization Rule to each word.
3. Until no more edges are added:
    a) Apply the Bottom-Up Predict Rule everywhere it applies.
    b) Apply the Fundamental Rule everywhere it applies.
1. Return all of the parse trees corresponding to the parse edges in the chart.

For example, the following diagram shows the order in which get added when applying bottom-up parsing to a simple example sentence:



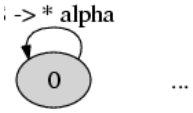
8.2.3 Top-Down Parsing

To create a bottom-up parser, we need to use the Fundamental Rule 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 captures the fact that root of any parse must be the start symbol. It states that for every grammar production:

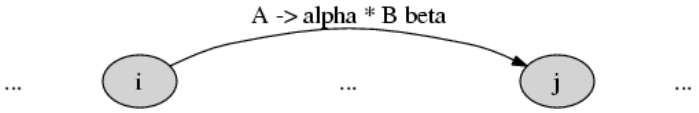
1. $S \rightarrow \alpha$

The parser should add the self-loop edge:



2. $[S \rightarrow \bullet\ \alpha\ ]@[0:0]$

The top-down expand rule says that if the 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**. In particular, if the chart contains the incomplete edge:



1. $[A \rightarrow \alpha \bullet B\ \beta\ ]@[i:j]$

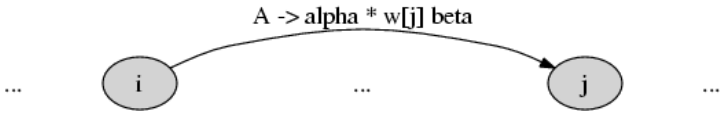
Then for each grammar production:

2. $B \rightarrow \gamma$

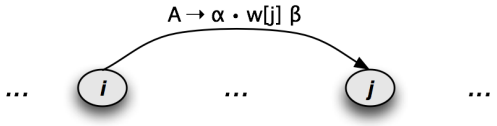
The parser should add the edge:

3. $[B \rightarrow \bullet\ \gamma\ ]@[j:j]$

The top-down match rule says that 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 text. In particular, if the chart contains the incomplete edge:



Alternative image:



1. $[A \rightarrow \alpha \bullet w_j\ \beta\ ]@[i:j]$

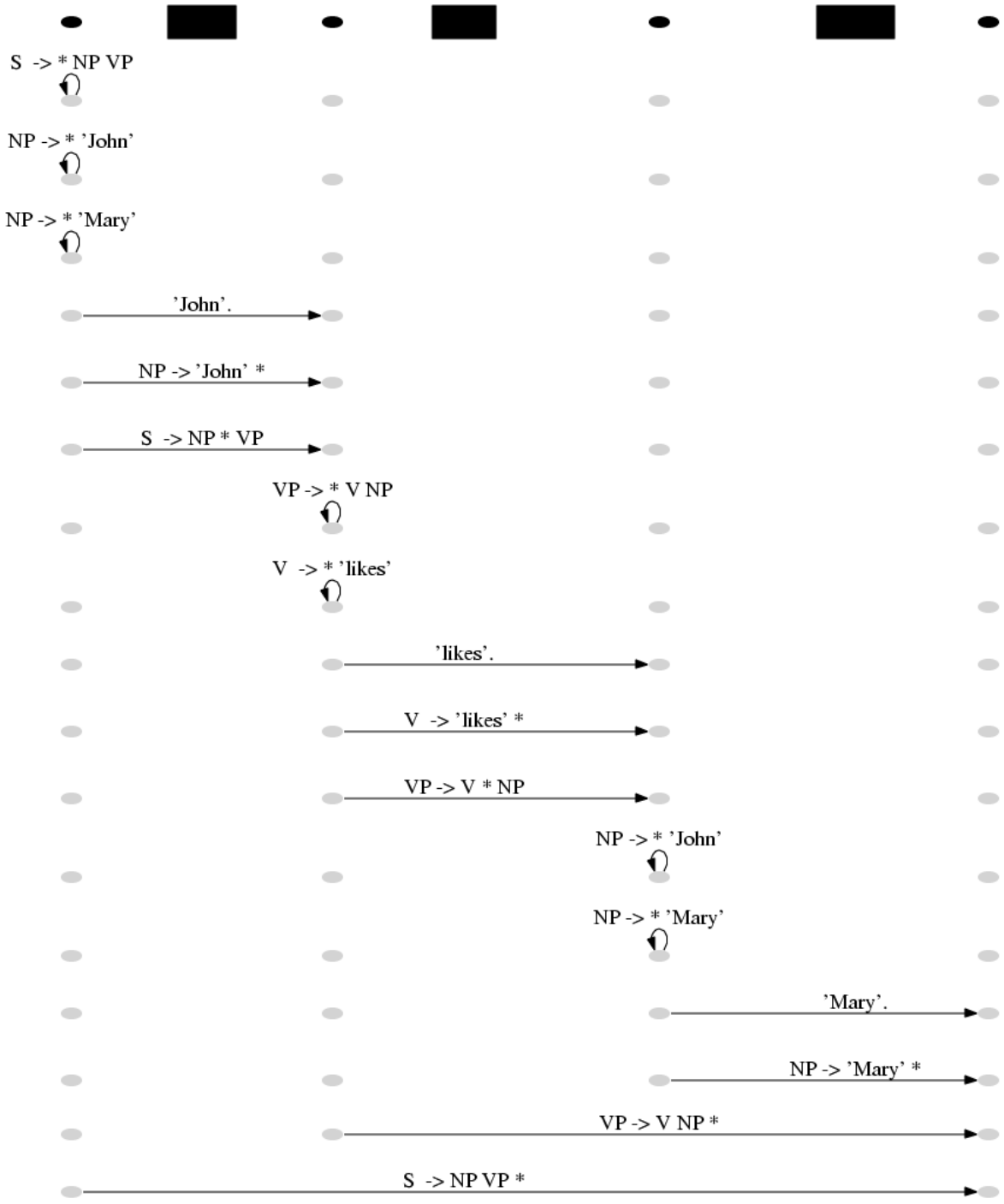
Then the parser should add the complete edge:

2. $[w_j; \rightarrow \bullet\ ]@[j:j+1]$

Using these four rules, we can parse a sentence as follows:

1. Create an empty chart spanning the sentence.
2. Apply the Top-Down Initialization Rule to each word.
3. Until no more edges are added:
    a) Apply the Top-Down Expand Rule everywhere it applies.
    b) Apply the Top-Down Match Rule everywhere it applies.
    c) Apply the Fundamental Rule everywhere it applies.
1. Return all of the parse trees corresponding to the parse edges in the chart.

For example, the following diagram shows the order in which get added when applying top-down parsing to a simple example sentence:



8.3 Chart Parsing in NLTK-Lite

8.3.1 Edges

NLTK defines two classes for encoding edges:

1. **`LeafEdge`** is used to encode edges of the form $[w_i \rightarrow \bullet\ ]@[i{:}i{+}1]$.

2. **TokenEdge** is used to encode edges of the form [A → α • β]@[i:j].

LeafEdges are constructed from a leaf terminal and an index:

```
>>> from nltk_lite.parse.chart import *
>>> edge2=LeafEdge('dog', 3)
>>> edge2
[Edge: [3:4] 'dog']
```

TreeEdges are constructed from a span, a left-hand side, a right-hand side, and a dot position:

```
>>> from nltk_lite.parse import cfg
>>> V, VP, NP, PP = cfg.nonterminals('V VP NP PP')
>>> edge1=TreeEdge((3,7), VP, [V,NP,PP], 2)
>>> edge1
[Edge: [3:7] VP -> V NP * PP]
```

The convenience function **TreeEdge.from_production** creates the **TreeEdge** licensed by a given CFG production:

```
>>> prod = cfg.parse_production('S->NP VP')[0]
>>> index = 3
>>> edge3 = TreeEdge.from_production(prod, 3)
>>> edge3
[Edge: [3:3] S  -> * NP VP]
```

Both **TreeEdge** and **LeafEdge** implement the **EdgeI** interface, which defines the methods that all edges should support:

The edge's span, start, end, and length >>> edge1.span() (3, 7) >>> edge1.start() 3 >>> edge1.end() 7 >>> edge1.length() 4

The edge's left-hand side and right-hand side. >>> edge1.lhs() <VP> >>> edge1.rhs() (<V>, <NP>, <PP>)

The edge's dot position. >>> edge1.dot() 2

Is it a complete edge? >>> edge1.is_complete() False >>> edge1.is_incomplete() True

The next RHS element after the dot >>> edge1.next() <PP> >>> edge2.next() None

8.3.2 Charts

Charts are encoded using the **Chart** class. To create an empty chart spanning a given sentence, use the **Chart** constructor:

```
>>> from nltk_lite import tokenize
>>> text = 'James wears a hat'
>>> chart = Chart(tokenize.whitespace(text))
```

New edges are added to the chart using the **insert** method, which takes an edge and a child pointer list. A *child pointer list* is a list of edges $e_1 \cdots e_d$, specifying the edges that licensed each child to the left of the dot. It is used to reconstruct the parse trees once parsing is finished.

```
>>> NP, N = cfg.nonterminals('NP N')
>>> edge1 = LeafEdge('hat', 3)
>>> edge2 = TreeEdge((3,3), NP, [N], 0)
>>> edge3 = TreeEdge((3,4), NP, [N], 1)
>>> chart.insert(edge1, [])
True
>>> chart.insert(edge2, [])
True
>>> chart.insert(edge3, [edge1])
True
```

Finally, we can pretty-print the chart with **chart.pp()** to get:

```
|.  James  .  wears  .    a    .   hat   .|
|.        .         .         >         .| [3:3] NP -> * N
|.        .         .         [---------]| [3:4] NP -> N *
|.        .         .         [---------]| [3:4] 'hat'
```

The leaves of the chart's token can be accessed with the methods **num_leaves**, **leaf**, and **leaves**. Note that this methods return the leaf properties of the words, and not the word tokens themselves:

```
>>> chart.num_leaves()
4
>>> chart.leaf(1)
'wears'
>>> chart.leaves()
['James', 'wears', 'a', 'hat']
```

The chart's edges can be accessed with the methods **num_edges** and **edges**:

```
>>> chart.num_edges()
3
>>> for edge in chart.edges(): print edge
[3:4] 'hat'
[3:3] NP -> * N
[3:4] NP -> N *
```

The **select** method can be used to efficiently retrieve all edges that satisfy one or more restrictions:

```
>>> for edge in chart.select(start=3): print edge
[3:4] 'hat'
[3:3] NP -> * N
[3:4] NP -> N *
>>> for edge in chart.select(lhs=NP): print edge
[3:3] NP -> * N
[3:4] NP -> N *
>>> for edge in chart.select(length=1): print edge
[3:4] 'hat'
[3:4] NP -> N *
>>> for edge in chart.select(lhs=NP, length=1): print edge
[3:4] NP -> N *
```

The following attributes can be given as restrictions to **select**: **span**, **start end**, **length**, **lhs**, **rhs**, **next**, **dot**, **is_complete**, **is_incomplete**.

The **trees** method returns a list of the trees that are associated with a given edge:

```
>>> chart.trees(edge3)
[(NP: 'hat')]
```

The **parses** method returns a list of the parse trees for a given start symbol. E.g. after having added many more edges, we could ask for the complete edges which span the entire chart, and which are based on a production from **S**, using **chart.parses('S')**:

(S: (NP: <James>) (VP: (V: <wears>) (NP: (Det: <a>) (N: <hat>))))

8.3.3 Chart Rules

The **ChartRuleI** class defines a standard interface for chart rules. Each chart rule must define the class variable **NUM_EDGES**, which specifies how many edges the rule applies to (e.g., two for the Fundamental Rule; one for the Top-Down Expand Rule; and none for the Top-Down Init Rule). Each chart rule must also define four methods:

1. **apply** adds all edges licensed by the rule and a given set of edges to the chart; and returns a list of the added edges.
2. **apply_everywhere** adds all edges licensed by the rule and the edges in the chart to the chart; and returns a list of the added edges.
3. **apply_iter** is a generator function that adds the edges licensed by the rule and a given set of edges to the chart, one at a time. Each time the generator is resumed, it adds a new edge and yields that edge; or returns.
4. **apply_everywhere_iter** is a generator function that adds the edges licensed by the rule and the edges in the chart to the chart, one at a time. Each time the generator is resumed, it adds a new edge and yields that edge; or returns.

To simplify chart rule construction, **nltk_lite.parse.chart** defines an abstract base class. **AbstractChartRule** provides default implementations for every method but **apply_iter**.

Currently, **nltk_lite.parse.chart** defines the following chart rules:

1. **FundamentalRule**: The Fundamental Rule.
2. **TopDownInitRule**: The Top Down Initialization Rule.
3. **TopDownExpandRule**: The Top Down Expand Rule.
4. **TopDownMatchRule**: The Top Down Match Rule.
5. **BottomUpInit**: The Bottom Up Initialization Rule.
6. **BottomUpPredictRule**: The Bottom Up Predict Rule.
7. **CachedTopDownInitRule**: A cached version of the Top Down Initialization Rule, to avoid recomputing edges for the same configuration
8. **CachedTopDownExpandRule**: A cached version of the Top Down Expand Rule, to avoid recomputing edges for the same configuration

9. **`SingleEdgeFundamentalRule`**: A single-edged version of the Fundamental Rule, that finds edges to combine with from the chart

10. **`CompleterRule`**: A single-edged version of **`FundamentalRule`** used by Earley's algorithm.

11. **`ScannerRule`**: A lexicon-based version of **`TopDownMatchRule`**, used by Earley's algorithm.

12. **`PredictorRule`**: Another name for **`TopDownExpandRule`**, used by Earley's algorithm.

8.3.4 ChartParser

`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, on order, until no new edges are added to the chart. In particular, **`ChartParse`** uses the following algorithm:

Until no new edges are added:

For each chart rule *$*:

Apply *R* to any applicable edges in the chart.

Return any complete parses in the chart.

`nltk_lite.parse.chart` defines two pre-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:

```
>>> grammar = cfg.parse_grammar('''
...   S -> NP VP
...   VP -> V NP | VP PP
...   V -> "saw" | "ate"
...   NP -> "John" | "Mary" | "Bob" | Det N | NP PP
...   Det -> "a" | "an" | "the" | "my"
...   N -> "dog" | "cat" | "cookie"
...   PP -> P NP
...   P -> "on" | "by" | "with"
...   ''')
```

Next we tokenize a sentence. We make sure it is a list (not an iterator), since we wish to use the same tokenized sentence several times.

```
>>> sent = list(tokenize.whitespace('John saw a cat with my cookie'))
>>> parser = ChartParse(grammar, BU_STRATEGY)
>>> for tree in parser.get_parse_list(sent):
...     print tree
(S:
  (NP: 'John')
  (VP:
```

```
    (VP: (V: 'saw') (NP: (Det: 'a') (N: 'cat')))
    (PP: (P: 'with') (NP: (Det: 'my') (N: 'cookie')))))
(S:
  (NP: 'John')
  (VP:
    (V: 'saw')
    (NP:
      (NP: (Det: 'a') (N: 'cat'))
      (PP: (P: 'with') (NP: (Det: 'my') (N: 'cookie'))))))
```

The **trace** parameter can be specified when creating a parser, to turn on tracing (higher trace levels produce more verbose output). The following examples show the trace output for parsing the same sentence with both the bottom-up and top-down strategies:

```
# Parse the sentence, bottom-up, with tracing turned on.
>>> parser = ChartParse(grammar, BU_STRATEGY, trace=2)
>>> parser.get_parse(sent)
|. John. saw .  a  . cat . with.  my .cooki.|
Bottom Up Init Rule:
|[-----]     .     .     .     .     .     .| [0:1] 'John'
|.     [-----]     .     .     .     .     .| [1:2] 'saw'
|.     .     [-----]     .     .     .     .| [2:3] 'a'
|.     .     .     [-----]     .     .     .| [3:4] 'cat'
|.     .     .     .     [-----]     .     .| [4:5] 'with'
|.     .     .     .     .     [-----]     .| [5:6] 'my'
|.     .     .     .     .     .     [-----]| [6:7] 'cookie'
Bottom Up Predict Rule:
|>     .     .     .     .     .     .     .| [0:0] NP -> * 'John'
|.     >     .     .     .     .     .     .| [1:1] V  -> * 'saw'
|.     .     >     .     .     .     .     .| [2:2] Det -> * 'a'
|.     .     .     >     .     .     .     .| [3:3] N  -> * 'cat'
|.     .     .     .     >     .     .     .| [4:4] P  -> * 'with'
|.     .     .     .     .     >     .     .| [5:5] Det -> * 'my'
|.     .     .     .     .     .     >     .| [6:6] N  -> * 'cookie'
Fundamental Rule:
|[-----]     .     .     .     .     .     .| [0:1] NP -> 'John' *
|.     [-----]     .     .     .     .     .| [1:2] V  -> 'saw' *
|.     .     [-----]     .     .     .     .| [2:3] Det -> 'a' *
|.     .     .     [-----]     .     .     .| [3:4] N  -> 'cat' *
|.     .     .     .     [-----]     .     .| [4:5] P  -> 'with' *
|.     .     .     .     .     [-----]     .| [5:6] Det -> 'my' *
|.     .     .     .     .     .     [-----]| [6:7] N  -> 'cookie' *
Bottom Up Predict Rule:
|>     .     .     .     .     .     .     .| [0:0] S  -> * NP VP
|>     .     .     .     .     .     .     .| [0:0] NP -> * NP PP
|.     >     .     .     .     .     .     .| [1:1] VP -> * V NP
|.     .     >     .     .     .     .     .| [2:2] NP -> * Det N
|.     .     .     .     >     .     .     .| [4:4] PP -> * P NP
|.     .     .     .     .     >     .     .| [5:5] NP -> * Det N
Fundamental Rule:
|[----->     .     .     .     .     .     .| [0:1] S  -> NP * VP
|[----->     .     .     .     .     .     .| [0:1] NP -> NP * PP
|.     [----->     .     .     .     .     .| [1:2] VP -> V * NP
```

```
|.     .     [----->     .     .     .     .| [2:3] NP -> Det * N
|.     .     [-----------]     .     .     .| [2:4] NP -> Det N *
|.     .     .     .     [----->     .     .| [4:5] PP -> P * NP
|.     .     .     .     .     [----->     .| [5:6] NP -> Det * N
|.     .     .     .     .     [-----------]| [5:7] NP -> Det N *
|.     [-----------------]     .     .     .| [1:4] VP -> V NP *
|.     .     .     .     [-----------------]| [4:7] PP -> P NP *
|[----------------------]     .     .     .| [0:4] S  -> NP VP *
Bottom Up Predict Rule:
|.     .     >     .     .     .     .     .| [2:2] S  -> * NP VP
|.     .     >     .     .     .     .     .| [2:2] NP -> * NP PP
|.     .     .     .     .     >     .     .| [5:5] S  -> * NP VP
|.     .     .     .     .     >     .     .| [5:5] NP -> * NP PP
|.     >     .     .     .     .     .     .| [1:1] VP -> * VP PP
Fundamental Rule:
|.     .     [----------->     .     .     .| [2:4] S  -> NP * VP
|.     .     [----------->     .     .     .| [2:4] NP -> NP * PP
|.     .     .     .     .     [-----------]| [5:7] S  -> NP * VP
|.     .     .     .     .     [-----------]| [5:7] NP -> NP * PP
|.     [-----------------]     .     .     .| [1:4] VP -> VP * PP
|.     .     [-----------------------------]| [2:7] NP -> NP PP *
|.     [-----------------------------------]| [1:7] VP -> VP PP *
|.     .     [----------------------------->| [2:7] S  -> NP * VP
|.     .     [----------------------------->| [2:7] NP -> NP * PP
|.     [-----------------------------------]| [1:7] VP -> VP * PP
|.     [-----------------------------------]| [1:7] VP -> V NP *
|[=======================================]| [0:7] S  -> NP VP *
|[=======================================]| [0:7] S  -> NP VP *
|.     [----------------------------------->| [1:7] VP -> VP * PP
(S: (NP: 'John') (VP: (VP: (V: 'saw') (NP: (Det: 'a') (N: 'cat'))) (PP: (P: 'with')
```

Next we parse the same sentence, top-down, with tracing turned on:

```
>>> parser = ChartParse(grammar, TD_STRATEGY, trace=2)
>>> parser.get_parse(sent)
|. John. saw .  a  . cat . with.  my .cooki.|
Top Down Init Rule:
|>     .     .     .     .     .     .     .| [0:0] S  -> * NP VP
Top Down Expand Rule:
|>     .     .     .     .     .     .     .| [0:0] NP -> * 'John'
|>     .     .     .     .     .     .     .| [0:0] NP -> * 'Mary'
|>     .     .     .     .     .     .     .| [0:0] NP -> * 'Bob'
|>     .     .     .     .     .     .     .| [0:0] NP -> * Det N
|>     .     .     .     .     .     .     .| [0:0] NP -> * NP PP
|>     .     .     .     .     .     .     .| [0:0] Det -> * 'a'
|>     .     .     .     .     .     .     .| [0:0] Det -> * 'an'
|>     .     .     .     .     .     .     .| [0:0] Det -> * 'the'
|>     .     .     .     .     .     .     .| [0:0] Det -> * 'my'
Top Down Match Rule:
|[-----]     .     .     .     .     .     .| [0:1] 'John'
Fundamental Rule:
|[-----]     .     .     .     .     .     .| [0:1] NP -> 'John' *
|[----->     .     .     .     .     .     .| [0:1] NP -> NP * PP
```

```
|[-----> .     .     .     .     .     .| [0:1] S  -> NP * VP
Top Down Expand Rule:
|.     > .     .     .     .     .     .| [1:1] PP -> * P NP
|.     > .     .     .     .     .     .| [1:1] VP -> * V NP
|.     > .     .     .     .     .     .| [1:1] VP -> * VP PP
|.     > .     .     .     .     .     .| [1:1] P  -> * 'on'
|.     > .     .     .     .     .     .| [1:1] P  -> * 'by'
|.     > .     .     .     .     .     .| [1:1] P  -> * 'with'
|.     > .     .     .     .     .     .| [1:1] V  -> * 'saw'
|.     > .     .     .     .     .     .| [1:1] V  -> * 'ate'
Top Down Match Rule:
|.     [-----] .     .     .     .     .| [1:2] 'saw'
Fundamental Rule:
|.     [-----] .     .     .     .     .| [1:2] V  -> 'saw' *
|.     [-----> .     .     .     .     .| [1:2] VP -> V * NP
Top Down Expand Rule:
|.     .     > .     .     .     .     .| [2:2] NP -> * 'John'
|.     .     > .     .     .     .     .| [2:2] NP -> * 'Mary'
|.     .     > .     .     .     .     .| [2:2] NP -> * 'Bob'
|.     .     > .     .     .     .     .| [2:2] NP -> * Det N
|.     .     > .     .     .     .     .| [2:2] NP -> * NP PP
|.     .     > .     .     .     .     .| [2:2] Det -> * 'a'
|.     .     > .     .     .     .     .| [2:2] Det -> * 'an'
|.     .     > .     .     .     .     .| [2:2] Det -> * 'the'
|.     .     > .     .     .     .     .| [2:2] Det -> * 'my'
Top Down Match Rule:
|.     .     [-----] .     .     .     .| [2:3] 'a'
Fundamental Rule:
|.     .     [-----] .     .     .     .| [2:3] Det -> 'a' *
|.     .     [-----> .     .     .     .| [2:3] NP -> Det * N
Top Down Expand Rule:
|.     .     .     > .     .     .     .| [3:3] N  -> * 'dog'
|.     .     .     > .     .     .     .| [3:3] N  -> * 'cat'
|.     .     .     > .     .     .     .| [3:3] N  -> * 'cookie'
Top Down Match Rule:
|.     .     .     [-----] .     .     .| [3:4] 'cat'
Fundamental Rule:
|.     .     .     [-----] .     .     .| [3:4] N  -> 'cat' *
|.     .     [-----------] .     .     .| [2:4] NP -> Det N *
|.     [-----------------] .     .     .| [1:4] VP -> V NP *
|.     .     [-----------> .     .     .| [2:4] NP -> NP * PP
|[----------------------] .     .     .| [0:4] S  -> NP VP *
|.     [-----------------> .     .     .| [1:4] VP -> VP * PP
Top Down Expand Rule:
|.     .     .     .     > .     .     .| [4:4] PP -> * P NP
|.     .     .     .     > .     .     .| [4:4] P  -> * 'on'
|.     .     .     .     > .     .     .| [4:4] P  -> * 'by'
|.     .     .     .     > .     .     .| [4:4] P  -> * 'with'
Top Down Match Rule:
|.     .     .     .     [-----] .     .| [4:5] 'with'
Fundamental Rule:
|.     .     .     .     [-----] .     .| [4:5] P  -> 'with' *
|.     .     .     .     [-----> .     .| [4:5] PP -> P * NP
```

```
Top Down Expand Rule:
|.     .     .     .     .     >     .     .| [5:5] NP -> * 'John'
|.     .     .     .     .     >     .     .| [5:5] NP -> * 'Mary'
|.     .     .     .     .     >     .     .| [5:5] NP -> * 'Bob'
|.     .     .     .     .     >     .     .| [5:5] NP -> * Det N
|.     .     .     .     .     >     .     .| [5:5] NP -> * NP PP
|.     .     .     .     .     >     .     .| [5:5] Det -> * 'a'
|.     .     .     .     .     >     .     .| [5:5] Det -> * 'an'
|.     .     .     .     .     >     .     .| [5:5] Det -> * 'the'
|.     .     .     .     .     >     .     .| [5:5] Det -> * 'my'
Top Down Match Rule:
|.     .     .     .     .     [-----]     .| [5:6] 'my'
Fundamental Rule:
|.     .     .     .     .     [-----]     .| [5:6] Det -> 'my' *
|.     .     .     .     .     [------>     .| [5:6] NP -> Det * N
Top Down Expand Rule:
|.     .     .     .     .     .     >     .| [6:6] N  -> * 'dog'
|.     .     .     .     .     .     >     .| [6:6] N  -> * 'cat'
|.     .     .     .     .     .     >     .| [6:6] N  -> * 'cookie'
Top Down Match Rule:
|.     .     .     .     .     .     [-----]| [6:7] 'cookie'
Fundamental Rule:
|.     .     .     .     .     .     [-----]| [6:7] N  -> 'cookie' *
|.     .     .     .     .     [-----------]| [5:7] NP -> Det N *
|.     .     .     .     [-----------------]| [4:7] PP -> P NP *
|.     .     .     .     .     [----------->| [5:7] NP -> NP * PP
|.     .     [-----------------------------]| [2:7] NP -> NP PP *
|.     [-------------------------------------]| [1:7] VP -> VP PP *
|.     [-------------------------------------]| [1:7] VP -> V NP *
|.     .     [----------------------------->| [2:7] NP -> NP * PP
|[=======================================]| [0:7] S  -> NP VP *
|.     [------------------------------------->| [1:7] VP -> VP * PP
|[=======================================]| [0:7] S  -> NP VP *
|.     [------------------------------------->| [1:7] VP -> VP * PP
Top Down Expand Rule:
|.     .     .     .     .     .     .     >| [7:7] PP -> * P NP
|.     .     .     .     .     .     .     >| [7:7] P  -> * 'on'
|.     .     .     .     .     .     .     >| [7:7] P  -> * 'by'
|.     .     .     .     .     .     .     >| [7:7] P  -> * 'with'
(S: (NP: 'John') (VP: (VP: (V: 'saw') (NP: (Det: 'a') (N: 'cat'))) (PP: (P: 'with')
```

8.4 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?