Handout 4: Chart parsing

1. Dotted rules

a. Keeping binary combination operation, while permitting rules with longer rhs's

b. Example: rule $X \to YZW$

c. Suppose the chart contains nodes $[_2Y_4]$, $[_4Z_7]$, $[_7W_8]$

start:			(α)	$(X \to\ {}_2Y_4 \bullet Z\ W)$
combine:	$(\alpha) + [_4Z_7]$	$\Rightarrow$	(β)	$(X \to\ {}_2Y_4\ {}_4Z_7 \bullet W)$
combine:	$(\beta) + [_7W_8]$	$\Rightarrow$	(γ)	$(X \to\ {}_2Y_4\ {}_4Z_7\ {}_7W_8 \bullet)$
complete:	(γ)	$\Rightarrow$		$[_2X_8]$

2. Two basic data structures

a. **Node** $[_2X_8]$

b. **Edge** $(X \to\ {}_2Y_4 \bullet Z\ W)$

3. Permits unary expansions, too

Rule	$X \to Y$
start:	$(X \to\ {}_2Y_4 \bullet)$
complete:	$[_2X_4]$

4. Example: figure 1

5. Operations

a. **Shift** each word$[i]$ with pos X to create node $[_iX_{i+1}]$

b. When a new node Y is created, **start** an edge for every rule $X \to Y \ldots$

c. **Combine** every edge $(\ldots_k \bullet Z \ldots)$ and node $[_kZ_j]$ to create edge $(\ldots\ {}_kZ_j \bullet \ldots)$

d. **Complete** each edge $(X \to\ {}_i \ldots {}_j \bullet)$ to create node $[_iX_j]$

6. Assuring systematicity

a. Keep a chart of nodes, do not create duplicates ${}_iX_j$

b. Shifts are no problem: once for each pos of each word

c. Starts are no problem: when we add a new node to the chart (as opposed to re-using an old node), immediately do all the starts for that node.

d. Completions are no problem: when the dot reaches the end, immediately create node ${}_iX_j$. If it already exists, add a new expansion rather than creating a new node.

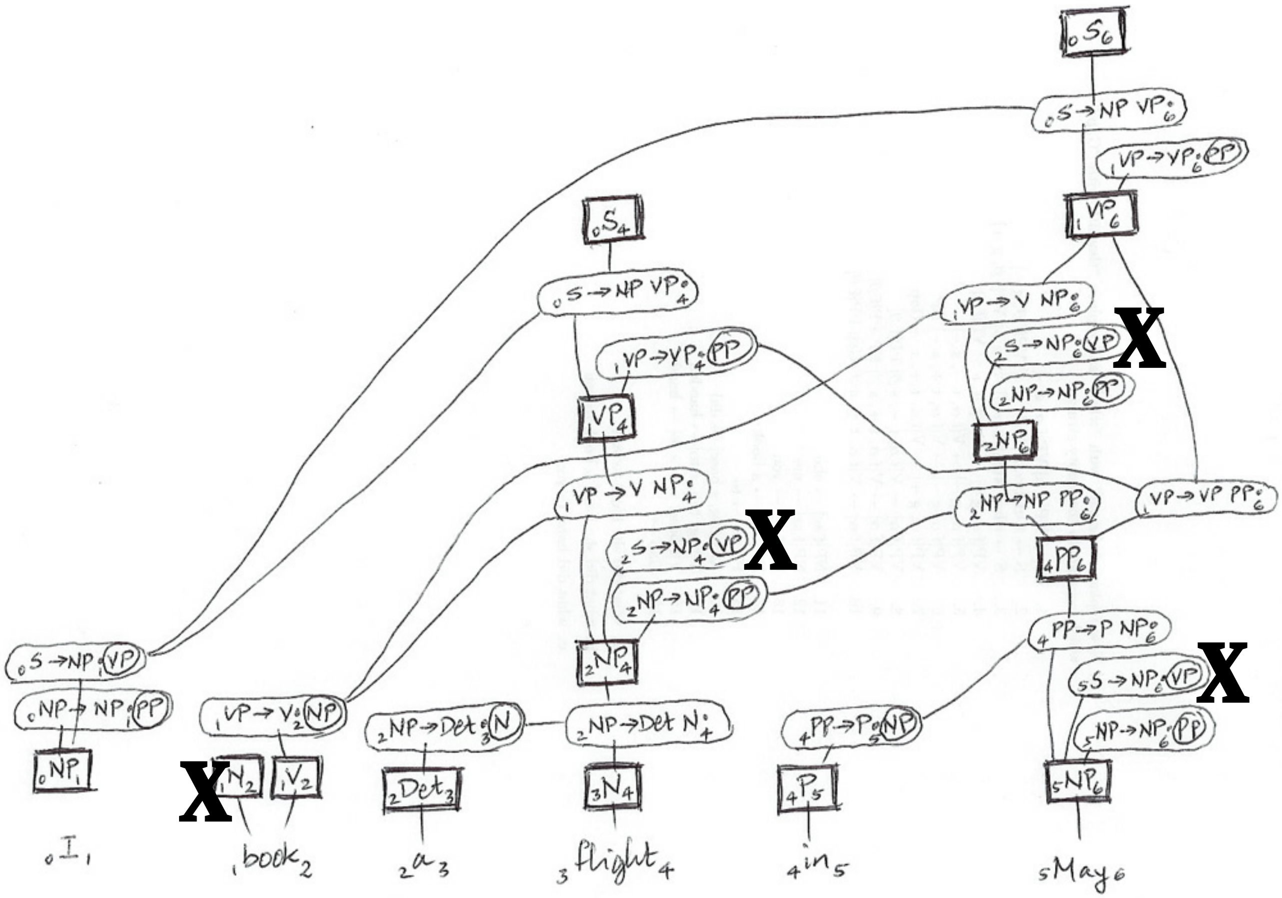

Figure 1: Filled chart for sentence "I book a flight in May." The node and edges marked with "X" are filtered out when topdown prediction is used. (Not shown is an initial prediction $\to \bullet_0 S$.)

e. Combinations—need to make sure all combinations get done exactly once.

7. Combinations

   a. Involve an edge $(\ldots_k \bullet Z \ldots)$ ending at k, and a node $[_k Z_j]$ beginning at k

   b. The danger: we combine all the edges *we know about* ending at k with nodes beginning at k, but later another edge gets created "behind our back," never gets combined with nodes

   c. The strategy: outer loop iterates through j from 1 to n.
      - Create edges $(\ldots_j \ \bullet \ldots)$ at time j
      - Create nodes $[_i X_j]$ at time j

   d. At time j, after creating node ${}_k Z_j$, we know that all edges $(\ldots_k \bullet Z \ldots)$ already exist, because $k < j$ and they were created at time k

8. Detail

   a. **shift**(j). For each part of speech X of the word w at $j-1$, do **add node**$(X, w, j-1, j)$.

   b. **start**(n), where n is a node of category Y. For each rule of form $X \to Y \ldots$, create an edge with partial expansion $[n]$, and do **add edge**.

   c. **combine**(n), where n is a node of form ${}_k Z_j$. For each old edge at k with Z after the dot, create a new edge in which n has been added to the expansion, and do **add edge**.

   d. **complete**(e), where e is an edge of form $(X \to \ {}_i \ldots_j \bullet)$. Do **add node**$(X, \ldots, i, j)$.

9. The glue

   a. **add node**(X, α, i, j), where α is either a word or a list of nodes. If there is already a node (X, i, j) in the chart, add α to its list of expansions. Otherwise, create a new node, and call **start** and **combine** on it.

   b. **add edge**(e), where e is a newly created edge. If the dot is at the end, call **complete**. Otherwise, e is of form $(X \to \ldots_k \bullet Y \ldots)$; add e to the edge table at index (k, Y).

10. The main loop

   a. Let j range from 1 to the length of the sentence; do **shift**(j)

   b. Just doing **shift**(j) starts a cascade: it calls **add node**, which calls **combine** and **start**. Both **combine** and **start** call **add edge**, which may call **complete**, which may call **add node** again.

c. After going through the sentence, if there is a node with category S, spanning the full sentence, unwind that node and return the resulting list of trees.

11. All functions should actually be methods of a `Parser` class

12. Nodes

- **a.** Attributes: `cat`, `expansions`, `i`, `j`.
- **b.** `Expansions` is a list whose elements are either strings (words) or node-lists.
- **c.** Creator `Node(cat,expansion,i,j)`
- **d.** Method `add(expansion)` just appends to the `expansions` list

13. Edges

- **a.** Attributes: `rule`, `expansion`. The expansion is a list of nodes.
- **b.** The dot is implicit: its position is determined by the length of the expansion.
- **c.** Creator: `Edge(rule, expansion)`
- **d.** Method `cat()` returns the lhs of the rule
- **e.** Method `start()` returns the start position of the first node, and `end()` returns the end position of the last node (just before the dot)
- **f.** Method `__add__()` takes a node and creates a new edge with an extended expansion
- **g.** Method `afterdot()` returns the next element in the rule's rhs, if the expansion is shorter than the rule rhs; and it returns `None` otherwise.

14. The chart

- **a.** It's just a `dict` (unique value for a given key)
- **b.** Keys are triples `(cat,i,j)`; values are nodes

15. The table of edges

- **a.** It's just an `Index` (list of values for a given key)
- **b.** Keys are pairs `(i,afterdot)`; values are edges

16. Unwinding is a bit tricky

a. Consider a parent node with category X and node-expansion `[child1, child2]`

b. Each node corresponds to multiple trees. Suppose `child1.trees()` is `[t1,t2]` and `child2.trees()` is `[u1,u2]`.

c. Each way of combining a `child1` tree t with a `child2` tree u gives us a **tree-expansion** $[t, u]$. A tree-expansion is a list of *trees,* not nodes.

d. Each tree-expansion $[t, u]$ gives us a tree $[_X\ t\ u]$ corresponding to the parent node.

e. When the parent node has multiple node-expansions, we simply pool all the trees coming from each.

17. The key step is (c)

a. Consider `[t1,t2]` and `[u1,u2]`. The tree-expansions are the **cross product**: `(t1,u1), (t1,u2), (t2, u1), (t2, u2)`

b. Define a function `cross_product()` that behaves like this:

```
>>> cross_product([['a','b'], [1,2]])
[('a', 1), ('a', 2), ('b', 1), ('b', 2)]
```

c. Then:

```
def tree_expansions (node_expansion):
    choices = [n.trees() for n in node_expansion]
    return cross_product(choices)
```

18. The `trees()` method of `Node`:

```
def trees (self):
    out = []
    for e in self.expansions:
        if isinstance(e, str):
            out.append(Tree(self.cat, word=e))
        elif len(e) == 0:
            out.append(Tree(self.cat))
        else:
            for childlist in tree_expansions(e):
                out.append(Tree(self.cat, childlist))
    return out
```

Modules

19. Finding and loading modules

a. If you start Python at the command line, and `tree.py` is in the local directory, you can do

```
>>> import tree
```

b. The environment variable `PYTHONPATH` is a colon-separate list of places to look for modules:

```
$ PYTHONPATH=/Users/me/python-stuff:$PYTHONPATH
$ export PYTHONPATH
```

c. In Python, you can add a directory to `sys.path`:

```
>>> import sys
>>> sys.path = ['/Users/me/python-stuff'] + sys.path
>>> import tree
```

d. If there's any confusion about which module you're getting:

```
>>> tree
<module 'tree' from '/Users/me/python-stuff/tree.pyc'>
>>> tree.__file__
'/Users/me/python-stuff/tree.pyc'
```