Lecture 14: Parsing part 2—Chart parsing

Henry S. Thompson
Drawing on slides by Mark Steedman via Philipp Koehn
18 October 2016

1. Recursive Descent Parsing: preliminaries

We're trying to build a parse tree, given

- a grammar
- an input, i.e. a sequence of terminal symbols

As for any other depth-first search, we may have to **backtrack**

- So we must keep track of **backtrack points**
- And whenever we make a **choice** among several rules to try, we add a backtrack point consisting of
  - a partial tree
  - the remaining as-yet-unexplored rules
  - and the as-yet-unconsumed items of input

Note that, to make the search go depth-first

- We'll use a **stack** to keep track
- That is, we'll operate **last in, first out** (LIFO)

Finally, we'll need a notion of where the focus of attention is in the tree we're building

- We'll call this the **subgoal**

2. Recursive Descent Parsing: Algorithm sketch

We start with

- a tree consisting of an 'S' node
  - with no children
  - This node is currently the subgoal
- An empty stack
- An input sequence
Repeatedly

1. If the subgoal is a non-terminal
   a. Choose a rule from the set of rules in the grammar whose left-hand sides match the subgoal
      a. For example, the very first time around the loop, we might choose
      b. $S \rightarrow NP \, VP$
      b. add children to the subgoal node corresponding to the symbols in the right-hand side of the chosen rule, in order
      a. In our example, that's two children, $NP$ and $VP$
   c. Make the first of these the new subgoal
      a. This is the other thing which makes this a depth-first search
   d. Go back to (1)
2. Otherwise (the subgoal is a terminal)
   a. If the input is empty, **Backtrack**
   b. If the subgoal matches the first item of input
      i. **Consume** the first item of the input
      ii. **Advance** the subgoal
      iii. Go back to (1)
   c. Otherwise (they don't match), **Backtrack**

3. Recursive Descent Parsing: Algorithm sketch, concluded

The three imperative actions in the preceding algorithm are defined as follows:

**Choose**

Pick one member from the set of rules

1. If the set has only one member, you're done
2. Otherwise, **push** a new backtrack point onto the stack
   ◦ With the unchosen rules, the current tree and subgoal and the current (unconsumed) input sequence

**Advance**

Change the subgoal, as follows:

1. If the current subgoal has a sibling to its right, pick that
2. Failing which, if the current subgoal is not the root, set the subgoal to the current subgoal's parent, and go back to (1)
3. Failing which, if the input is empty, we win
   ◦ The current subgoal is the 'S' at the root, and it is the top node of a complete parse tree for the original input
4. Otherwise, **Backtrack**

**Backtrack**

Try to, as it were, change your mind. That is:

1. Unless the stack is empty, **pop** the top backtrack point off the backtrack stack and
   a. Set the tree, subgoal and input from it
   b. **Choose** a rule from its set of rules
   c. Go back to step (1b) of the algorithm
2. Otherwise (the stack is empty)
   ◦ We lose!
   ◦ There is no parse for the input with the grammar
4. Search Strategies

Schematic view of the search space:

In **depth-first** search the parser

- explores one branch of the search space at a time
- If this branch is a dead-end, it needs to **backtrack**

In **breadth-first** search the parser

- explores all possible branches in parallel
  - often rendered impossible by memory requirements

5. Shift-Reduce Parsing

**Search strategy** does not imply a particular **directionality** in which structures are built.

Recursive descent parsing **searches depth-first** and **builds top-down**.

Although **Shift-reduce parsing** also **searches depth-first**, in contrast it **builds structures bottom-up**.

It does this by repeatedly

1. **shifting** terminal symbols from the input string onto a stack
2. **reducing** some elements of the stack to the LHS side of a rule when they match its RHS

As described, this is just a recogniser

- You win if you end up with a single 'S' on the stack and no more input

Actual **parsing** requires more bookkeeping

Given certain constraints, it is possible to pre-compute auxiliary information about the grammar and exploit it during parsing so that no backtracking is required.

Modern computer languages are often parsed this way

- But grammars for natural languages don't (usually) satisfy the relevant constraints
6. Global and Local Ambiguity

A string can have more than one structural analysis (called **global ambiguity**) for one or both of two reasons:

- Grammatical rules allow for different attachment options;
- Lexical rules that allow a word to be in more than one word class.

Within a single analysis, some sub-strings can be analysed in more than one way

- even if not all these sub-string analyses 'survive'
- That is, if they are not compatible with any complete analysis of the entire string
- This is called **local ambiguity**

Local ambiguity is very common in natural languages as described by formal grammars

All depth-first parsing is inherently **serial**, and serial parsers can be massively inefficient when faced with **local ambiguity**.

7. Complexity

Depth-first parsing strategies demonstrate other problems with "parsing as search":

1. **Structural ambiguity** in the grammar and **lexical ambiguity** in the words (that is, words occurring under more than one part of speech) may lead the parser down a wrong path
2. So the same sub-tree may be built several times
   - whenever a path fails, the parser abandons any subtrees computed since the last backtrack point, backtracks and starts again

The complexity of this **blind backtracking** is exponential in the worst case because of repeated **re-analysis** of the same sub-string.

[Worked example on the whiteboard, for "gave a book to Robin" with this grammar:

```
VP → Vditr NP NP  NP → Det N
VP → Vtrpp NP PP  NP → PropN
PP → P NP
```

**Chart parsing** is the name given to a family of solutions to this problem

8. Dynamic Programming

It seems like we should be able to avoid the kind of repeated reparsing a simple recursive descent parser must often do

A CFG parser, that is, a **context-free parser**, should be able to avoid re-analyzing sub-strings

- because the analysis of any sub-string is **independent** of the rest of the parse
The parser’s exploration of its search space can exploit this independence

- if the parser uses **dynamic programming**.

Dynamic programming is the basis for all **chart parsing** algorithms.

### 9. Parsing as dynamic programming

Given a problem, **dynamic programming** systematically fills a table of solutions to sub-problems

- A process sometimes called **memoisation**

Once solutions to all sub-problems have been accumulated

- DP solves the overall problem by composing them

For parsing, sub-problems are analyses of sub-strings

- which can be memoised
- in a **chart**
- also know as a **well-formed substring table**, WFST

Each entry in the chart or WFST corresponds to either:

- a complete **constituent** (sub-tree), indexed by the start and end of the sub-string that it covers;
- or a **hypothesis** about what complete constituent might be found, indexed by the start and end of any constituent sub-strings found so far

### 10. Depicting a WFST/Chart

A well-formed substring table (aka chart) can be depicted as either a **matrix** or a **graph**

- Both contain the same information

When a **WFST** (aka **chart**) is depicted as a matrix:

- Rows and columns of the matrix correspond to the start and end positions of a span of items from the input
  - That is, starting **right before** the first word, ending **right after** the final one
- A cell in the matrix corresponds to the sub-string that starts at the row index and ends at the column index
- A cell can contain
  - information about the **type** of constituent (or constituents) that span(s) the substring
  - pointers to its sub-constituents
  - if the constituent is incomplete, **predictions** about what constituents might follow the substring
### 11. Depicting a WFST as a matrix

Here's a sample matrix, part-way through a parse:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Prep</td>
<td></td>
<td>PP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Det</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 **See 1 with 2 a 3 telescope 4 in 5 hand 6**

We can read this as saying:

- There is a **PP** from 1 to 4
  - Because there is a **Prep** from 1 to 2
  - and an **NP** from 2 to 4

### 12. Algorithms for chart parsing

Important examples of parser types which use a WFST include:

- The CKY algorithm, which memoises only complete constituents
- Three algorithm families that involve memoisation of both complete and incomplete constituents
  - Incomplete constituents can be understood as **predictions**
    - bottom-up chart parsers
    - May include top-down filtering
    - top-down chart parsers
      - May include bottom-up filtering
    - the Earley algorithm

### 13. CKY Algorithm

CKY (Cocke, Kasami, Younger) is an algorithm for recognising constituents and recording them in the chart (WFST).
CKY was originally defined for Chomsky Normal Form

\[
\begin{align*}
A & \rightarrow BC \\
A & \rightarrow a
\end{align*}
\]

- (Much more recently, this restriction has been lifted in a version by Lang and Leiss)

We can enter constituent \( A \) in cell \((i,j)\) iff either

- there is a rule \( A \rightarrow b \) and
  - \( b \) is found in cell \((i,j)\)
- or if there is a rule \( A \rightarrow B \ C \) and there is at least one \( k \) between \( i \) and \( j \) such that
  - \( B \) is found in cell \((i,k)\)
  - \( C \) is found in cell \((k,j)\)

14. CKY parsing, cont'd

Proceeding systematically bottom-up, CKY guarantees that the parser only looks for rules which might yield a constituent from \( i \) to \( j \) after it has found all the constituents that might contribute to it, that is

- That are shorter than it is
- That end at or to the left of \( j \)
- This guarantees that every possible constituent will be found

Note that this process manifests the fundamental weakness of blind bottom-up parsing:

- Large numbers of constituents will be found which do not participate in the ultimately spanning 'correct' analyses.
15. Visualising the chart: YACFG

Grammatical rules
- S → NP VP
- NP → Det Nom
- Nom → N SRel
- Nom → N
- VP → TV NP
- VP → IV PP
- VP → IV
- PP → Prep NP
- SRel → Relpro VP

Lexical rules
- Det → a | the (determiner)
- N → fish | frogs | soup (noun)
- Prep → in | for (preposition)
- TV → saw | ate (transitive verb)
- IV → fish | swim (intransitive verb)
- Relpro → that (relative pronoun)

Nom: nominal (the part of the NP after the determiner, if any)
SRel: subject relative clause, as in the frogs that ate fish.

Non-terminals occurring (only) on the LHS of lexical rules are sometimes called pre-terminals
- In the above grammar, these are Det, N, Prep, TV, IV, Relpro

Sometimes instead of sequences of words
- we just parse sequences of pre-terminals
- At least during grammar development

16. Visualising the chart: the setup

Just the empty matrix

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

the    frogs    ate    fish
### 17. Visualising the Chart (0,1)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the  frogs  ate  fish

Unary branching rules: det → the

### 18. Visualising the Chart (1,2)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>n</td>
<td>nom</td>
<td>np</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the  frogs  ate  fish
Unary branching rules: N → frogs, Nom → N, NP → Nom

19. Visualising the Chart (2,3)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>n nom np</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>tv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>the frogs</td>
<td>ate</td>
<td>fish</td>
<td></td>
</tr>
</tbody>
</table>

Unary branching rules: tv → ate
20. Visualising the Chart (3,4)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>n</td>
<td>nom</td>
<td>np</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>tv</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nom</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>np</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vp</td>
</tr>
</tbody>
</table>

the frogs ate fish

Unary branching rules: N → fish, Nom → N, NP → Nom, iv → fish, vp → iv

21. Visualising the Chart (0,2)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td>np</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>n</td>
<td>nom</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>np</td>
<td>np</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>tv</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nom</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>np</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vp</td>
</tr>
</tbody>
</table>

the frogs ate fish
Binary branching rule: NP → Det Nom

- $(0,1) \& (1,2) \Rightarrow (0,2)$

### 22. Visualising the Chart (1,3)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td>np</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>nom</td>
<td>np</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>tv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>n</td>
<td>nom</td>
</tr>
</tbody>
</table>

- $(1,2) \& (2,3) \not\Rightarrow$

---

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>frogs</th>
<th>ate</th>
<th>fish</th>
</tr>
</thead>
</table>

- $(1,2) \& (2,3) \not\Rightarrow$
23. Visualising the Chart (2,4)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td>np</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>nom</td>
<td>np</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>tv</td>
<td>vp</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>nom</td>
<td>np</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>iv</td>
<td>vp</td>
</tr>
</tbody>
</table>

the  | frogs | ate | fish

Binary branching rule: $VP \rightarrow TV\ NP$

- $(2,3) \& (3,4) \Rightarrow (2,4)$
24. Visualising the Chart (1,4)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td>np</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>nom</td>
<td></td>
<td>s</td>
</tr>
<tr>
<td></td>
<td>np</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>tv</td>
<td>vp</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>nom</td>
<td>np</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iv</td>
<td>vp</td>
<td></td>
</tr>
</tbody>
</table>

the frogs ate fish

Binary branching rule: $S \rightarrow NP \ VP$

- $(1,2) \& (2,4) \Rightarrow (1,4)$
- $(1,3) \& (3,4) \nRightarrow$
25. Visualising the Chart (0,4)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td>np</td>
<td></td>
<td>s</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>np</td>
<td>nom</td>
<td>s</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>tv</td>
<td></td>
<td>vp</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nom</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>np</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vp</td>
</tr>
<tr>
<td>the</td>
<td>frogs</td>
<td>ate</td>
<td>fish</td>
<td></td>
</tr>
</tbody>
</table>

Binary branching rule: $S \rightarrow NP \ VP$

- $(0,1) \& (1,4) \rightarrow$
- $(0,2) \& (2,4) \Rightarrow (0,4)$
- $(0,3) \& (3,4) \rightarrow$

26. From CKY Recogniser to CKY Parser

We cannot tell from the CKY chart as specified, the syntactic analysis of the input string.

We just have a chart recogniser, a way of determining whether a string belongs to the language generated by the grammar.

Changing this to a parser requires recording which existing constituents were combined to make each new constituent. This requires another field to record the one or more ways in which a constituent spanning $(i,j)$ can be made from constituents spanning $(i,k)$ and $(k,j)$. 