Introduction to Computational Linguistics: Parsing Algorithms

Sharon Goldwater
(based on slides by Mark Steedman and Philipp Koehn)

20 July 2015

Parsers

A **parser** is an algorithm that computes a structure for an input string given a grammar.

Understanding different parsing algorithms is important for:

- Computer scientists: parsers used to compile programs, check html, etc.
- NLP researchers: efficient parsers needed for large-scale language tasks (e.g., used to create Google's "infoboxes").
- Psycholinguists: what algorithm might be used by the human sentence processing mechanism?

Ambiguity refresher

Parsers need to handle (rampant!) syntactic ambiguity in natural language.

- **global ambiguity**: multiple full parses are possible, e.g., PP attachment: I saw the man with the telescope
- **local ambiguity**: ambiguous partial structures, need not be consistent with full parse.
  - classic garden path sentences: the old man the boats
  - but also lots of "normal" sentences: the dog bit the child
- Ambiguity can be **structural** (different possible phrasal constituents) or **lexical** (word with multiple POS tags), often both.

CFG refresher

Parsing algorithms exist for many types of grammars, but we'll consider just context-free grammars for now. CFG refresher (or see J&M 12.2-12.5):

- Two types of grammar symbols:
  - **terminals** (t): words.
  - **Non-terminals** (NT): phrasal categories like S, NP, VP, PP. Sometimes we distinguish **pre-terminals** (POS tags), a type of NT.
- Rules must have the form $\mathtt{NT} \rightarrow \beta$, where β is any string of NT's and t's.
- A CFG in **Chomsky Normal Form** only has rules of the form $\mathtt{NT}_i \rightarrow \mathtt{NT}_j\ \mathtt{NT}_k$ or $\mathtt{NT}_i \rightarrow \mathtt{t}_j$

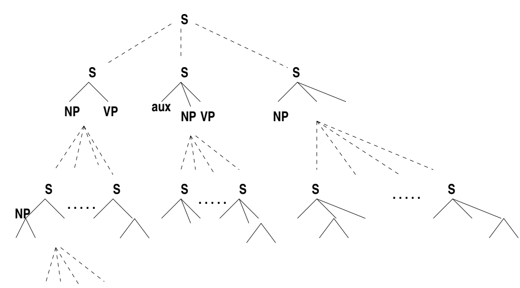
Parser properties

All parsers have two fundamental properties:

- **Directionality**: the sequence in which the structures are constructed.
  - **top-down**: start with root category (S), choose expansions, build down to words.
  - **bottom-up**: build subtrees over words, build up to S.
  - **Mixed** strategies also possible (e.g., left corner parsers)
- **Search strategy**: the order in which the search space of possible analyses is explored.

Example: search space for top-down parser

- Start with S node.
- Choose one of many possible expansions.
- Each of which has children with many possible expansions...
- etc



Search strategies

- **depth-first search**: explore one branch of the search space at a time, as far as possible. If this branch is a dead-end, parser needs to **backtrack.**
- **breadth-first search**: expand all possible branches in parallel (or simulated parallel). Requires storing many incomplete parses in memory at once.
- **best-first search**: score each partial parse and pursue the highest-scoring options first. (Will get back to this when discussing statistical parsing.)

Recursive Descent Parsing

- A **recursive descent** parser treats a grammar as a specification of how to break down a top-level goal (find S) into subgoals (find NP VP).
- It is a **top-down**, **depth-first** parser:
  - blindly expand nonterminals until reaching a terminal (word).
  - If terminal matches next input word, continue; else, backtrack.

Backtrack points

- Need to keep track of **backtrack points**, to return to if we backtrack.
- Each backtrack point stores:
  - A partial parse tree (what was completed when we made a choice)
  - The rules we haven't tried yet
  - The input words we haven't matched yet
- To ensure *depth-first* search, backtrack points are stored in a **stack**: last in, first out.

RD Parsing: initialization

We start with

- The rules of our context-free grammar, e.g.,

S → NP VP	VP → V	NN → bit	V → bit
NP → DT NN	DT → the	NN → dog	V → dog

- Current partial parse (also current subgoal): the S node.
- An ordered list of subgoals, initially containing just S.
- An empty stack of backtrack points.
- The input sequence (e.g., the dog bit)

RD Parsing: iterative steps

- If first subgoal in list is a *non-terminal* A:
  - Pick a rule from the grammar to expand it (e.g., A → B C)
  - Replace A in subgoal list with B C
- If first subgoal in list is a *terminal* w:
  - If input is empty, backtrack.
  - If next input word is different from w, backtrack.
  - If next input word is w, match! i.e., consume input word w and subgoal w and move to next subgoal.

RD Parsing: iterative steps

- If first subgoal in list is a *non-terminal* A:
  - Pick a rule from the grammar to expand it (e.g., A → B C)
  - Replace A in subgoal list with B C
- If first subgoal in list is a *terminal* w:
  - If input is empty, backtrack.*
  - If next input word is different from w, backtrack.
  - If next input word is w, match! i.e., consume input word w and subgoal w and move to next subgoal.**

* If stack is empty, we lose! No parse is possible.
** If no more subgoals, we win! We found a parse.

Recursive descent example

- Grammar and sentence from slide 9.
- Operations:
  - Expand (E)
  - Match (M)
  - Backtrack to step *n* (B*n*)

Step	Op.	Subgoals	Input
0		S	the dog bit
1	E	NP VP	the dog bit
2	E	DT NN VP	the dog bit
3	E	the NN VP	the dog bit
4	M	NN VP	dog bit
5	E	bit VP	dog bit
6	B4	NN VP	dog bit
7	E	dog VP	dog bit
8	M	VP	bit
9	E	V	bit
10	E	bit	bit
11	M		

Parsers vs Recognizers

- The above sketch is actually a **recognizer**: it tells us whether the sentence has a valid parse, but not what the parse is.
- Would need to add more details to keep track of parse structure as it is built.

Shift-Reduce Parsing

- Search strategy and directionality are orthogonal properties.
- **Shift-reduce** parsing is **depth-first** (like RD) but **bottom-up** (unlike RD).
- Basic shift-reduce recognizer repeatedly:
  - **Shifts** input symbols onto a stack.
  - Whenever possible, **reduces** one or more items from top of stack that match RHS of rule, replacing with LHS of rule.
- Like RD parser, needs to maintain backtrack points.

Shift-reduce example

- Same example grammar and sentence.
- Operations:
  - Shift (S)
  - Reduce (R)
  - Backtrack to step *n* (B*n*)

Step	Op.	Stack	Input
0			the dog bit
1	S	the	dog bit
2	R	DT	dog bit
3	S	DT dog	bit
4	R	DT V	bit
5	S	DT V bit	
6	R	DT V V	
7	B5	DT V bit	
8	R	DT V NN	
9	B3	DT dog	bit
10	R	DT NN	bit
11	R	NP	bit
12	R	NP bit	bit
...			

Depth-first parsing in practice

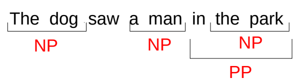
- Depth-first parsers are very efficient for unambiguous structures.
  - Widely used to parse/compile programming languages
  - Language/grammar is specially constructed to be unambiguous (sometimes with finite **lookahead**).

Depth-first parsing in practice

- Depth-first parsers are very efficient for unambiguous structures.
  - Widely used to parse/compile programming languages
  - Language/grammar is specially constructed to be unambiguous (sometimes with finite **lookahead**).
- But can be massively inefficient (exponential in sentence length) if faced with local ambiguity.
  - Blind backtracking may require re-building the same structure over and over.
  - So, much less common for natural language parsing (though some work on best-first probabilistic shift-reduce parsing: uses lookahead to help predict which expansions to make).

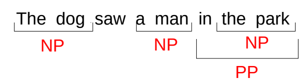
Breadth-first search using dynamic programming

- With a CFG, a 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.



Breadth-first search using dynamic programming

- With a CFG, a 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.



- To exploit this fact, **chart parsing** algorithms use dynamic programming to store and reuse sub-parses.
- This permits exploring multiple potential parses at once: a breadth-first strategy.

Parsing as dynamic programming

- As in HMM algorithms, dynamic programming fills a table of solutions to subproblems (memoization), then composes these to find the full solution.
- For parsing, subproblems are analyses of substrings, memoized in **chart** (aka **well-formed substring table**, WFST).
- Chart entries are indexed by *start* and *end* positions in the sentence, and correspond to:
  - either a complete **constituent** (sub-tree) spanning those positions (if working bottom-up),
  - **or** a **hypothesis** about what complete constituent might be found (if working top-down).

Depicting a WFST/Chart

- Chart can be depicted as either a **matrix** or a **graph**.
- In either case, we assume **indices** between each word in the sentence:

$_0$ See $_1$ with $_2$ a $_3$ telescope $_4$ in $_5$ hand $_6$

- If using a matrix, cell $[i, j]$ holds information about the word span from position i to position j:
  - The root node of any constituent(s) spanning those words
  - Pointers to its sub-constituents
  - (Depending on parsing method,) **predictions** about what constituents might follow the substring.

Depicting a WFST as a Matrix

Example, partway through parsing.

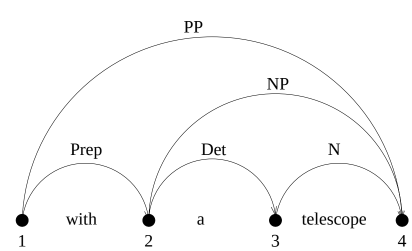
- Here, only showing root nodes of each constituent.
- Lower left of chart never used; often only upper right is shown.

	1	2	3	4	5	6
0	V					
1		Prep		PP		
2			Det	NP		
3				N		
4						
5						

$_0$ See $_1$ with $_2$ a $_3$ telescope $_4$ in $_5$ hand $_6$

Depicting a WFST as a Graph

- Here, each sentence position index is a **node** or **vertex**.
- **edges** (arcs) represent spans, labelled with the same information that goes in a cell in the matrix representation.



Algorithms for Chart Parsing

Many different chart parsing algorithms, including

- the **CKY algorithm**, which memoizes only complete constituents
- various algorithms that also memoize predictions/partial constituents
  - often using mixed bottom-up and top-down approaches, e.g., the Earley algorithm described in J&M, and left-corner parsing.

CKY Algorithm

CKY (Cocke, Kasami, Younger) is a **bottom-up**, **breadth-first** parsing algorithm.

- Original (simplest) version assumes grammar in Chomsky Normal Form.
- Add constituent `A` in cell (i, j) if:
  - there is a rule `A` $\rightarrow$ `B`, and `B` is in cell (i, j), **or**
  - there is a rule `A` $\rightarrow$ `B C`, and `B` is in cell (i,k) and `C` is in cell (k, j).

CKY Algorithm

CKY (Cocke, Kasami, Younger) is a **bottom-up**, **breadth-first** parsing algorithm.

- Original (simplest) version assumes grammar in Chomsky Normal Form.
- Add constituent `A` in cell (i, j) if:
  - there is a rule `A` $\rightarrow$ `B`, and `B` is in cell (i, j), **or**
  - there is a rule `A` $\rightarrow$ `B C`, and `B` is in cell (i,k) and `C` is in cell (k, j).
- Fills chart in order: only looks for rules that use a constituent from i to j **after** finding all constituents ending at i. So, guaranteed to find all possible parses.

CKY Pseudocode

- Assume input sentence with indices 0 to n, and chart c.

```
for len = 1 to n:   #number of words in constituent
   for i = 0 to n-len:   #start position
      j = i+len   #end position
      #process unary rules
      if A->B and c[i,j] has B, add A to c[i,j]
      for k = i+1 to j-1   #mid position
         #process binary rules
         if A->B C and c[i,k] has B and c[k,j] has C, add A to c[i,j
```

- This algorithm performs *recognition* in time $O(n^3)$.

CKY example

Grammatical rules	Lexical rules
S $\rightarrow$ NP VP	Det $\rightarrow$ a | the **(determiner)**
NP $\rightarrow$ Det Nom	N $\rightarrow$ fish | frogs | soup **(noun)**
NP $\rightarrow$ Nom	Prep $\rightarrow$ in | for **(preposition)**
Nom $\rightarrow$ N SRel	TV $\rightarrow$ saw | ate **(transitive verb)**
Nom $\rightarrow$ N	IV $\rightarrow$ fish | swim **(intransitive verb)**
VP $\rightarrow$ TV NP	Relpro $\rightarrow$ that **(relative pronoun)**
VP $\rightarrow$ IV PP	
VP $\rightarrow$ IV	
PP $\rightarrow$ Prep NP	
SRel $\rightarrow$ Relpro VP	

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

Visualizing the Chart

	1	2	3	4
0				
1				
2				
3				
	the	frogs	ate	fish

Visualizing the Chart (0,1)

	1	2	3	4
0	det			
1				
2				
3				
	the	frogs	ate	fish

Unary branching rules: det $\rightarrow$ the

Visualizing the Chart (1,2)

	1	2	3	4
0	det			
1		n nom np		
2				
3				
	the	frogs	ate	fish

Unary branching rules: N $\rightarrow$ frogs, Nom $\rightarrow$ N, NP $\rightarrow$ Nom

Visualizing the Chart (2,3)

	1	2	3	4
0	det			
1		n nom np		
2			tv	
3				
	the	frogs	ate	fish

Unary branching rules: tv $\rightarrow$ ate

Visualizing the Chart (3,4)

	1	2	3	4
0	det			
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

Unary branching rules: N $\rightarrow$ fish, Nom $\rightarrow$ N, NP $\rightarrow$ Nom, iv $\rightarrow$ fish, vp $\rightarrow$ iv

Visualizing the Chart (0,2)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: NP $\rightarrow$ Det Nom **(0,1) & (1,2) $\rightsquigarrow$ (0,2)**

Visualizing the Chart (1,3)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

(1,2) & (2,3) $\not\rightsquigarrow$

Visualizing the Chart (2,4)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: VP $\rightarrow$ TV NP **(2,3) & (3,4) $\rightsquigarrow$ (2,4)**

Visualizing the Chart (0,3)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

(0,1) & (1,3) $\not\rightsquigarrow$ **(0,2) & (2,3) $\not\rightsquigarrow$**

Visualizing the Chart (1,4)

	1	2	3	4
0	det	np		
1		n nom np		s
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

Binary rule: S $\rightarrow$ NP VP **(1,2) & (2,4) $\rightsquigarrow$ (1,4)** **(1,3) & (3,4) $\not\rightsquigarrow$**

Visualizing the Chart (0,4)

	1	2	3	4
0	det	np		s
1		n nom np		s
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

(0,1) & (1,4) $\not\rightsquigarrow$
Binary rule: S $\rightarrow$ NP VP **(0,2) & (2,4) $\rightsquigarrow$ (0,4)** **(0,3) & (3,4) $\not\rightsquigarrow$**

A note about CKY ordering

- Notice that to fill cell (i, j), we use a cell from row i and a cell from column j.
- So, we must fill in all cells down and left of (i, j) before filling (i, j).
- Here, we filled in all short entries, then longer ones, but other orders can work (e.g., J&M fill in all spans ending at j, then increment j.)

From CKY Recognizer to CKY Parser

- As just specified, CKY only *recognizes*, but can't return the parse.
- e.g., we don't know from S cell how it was constructed.
- Easy to fix:
  - whenever a constituent is found for cell (i, j), store the indices of the sub-constituents that formed it.
  - can mean storing multiple copies of A with different indices.
  - Sometimes called a **packed parse forest**: represents a possibly exponential number of trees in a compact way.

CKY in practice

- Avoids re-computing substructures, so much more efficient than depth-first parsers (in worst case).
- Still may compute a lot of unnecessary partial parses.
- Simple version requires converting the grammar to CNF (may cause blowup).

Various other chart parsing methods avoid these issues by combining top-down and bottom-up approaches (see J&M).

We also haven't said anything about how to choose between different parses when there's global ambiguity.

Introduction to Computational Linguistics: Parsing and Human Sentence Processing

Sharon Goldwater

20 July 2015

School of informatics

Recap: Parsing

We have seen several different parsing algorithms:

- Recursive descent parsing: top-down, depth-first.
- Shift-reduce parsing: bottom-up, depth-first.
- CKY parsing: bottom-up, breadth-first.

Do any of these seem plausbile as a model of *human* sentence processing?

Properties of human parsing mechanism

- **Fast**: real-time.
- **Recognizes global ambiguity**: at least to some extent.
- **Incremental**: words (and meaning) are integrated into structure immediately.
- **Can be led astray**: by local ambiguity (garden path sentences).
- **But mostly isn't**: many local ambiguities don't cause problems.

Global ambiguity

- Some examples are clearly (even humorously) ambiguous:
  - I saw the man with a telescope.
  - She sat on the chair covered in dust.
  - Milk drinkers are turning to powder.
- But most ambiguity isn't even noticed!
  - I saw the man with a hat.
  - She stood on the stoop covered in tears.
  - Breast feeders are turning to a new enriched formula.

Local ambiguity

- Same goes for local ambiguity:
  - The old man the boats
  - We painted the wall with cracks
  - Fat people eat accumulates

  versus

  - The dog bit the cat
  - The green is used for playing soccer
  - We stopped short of going

Properties of our parsing algorithms

- **Fast?** We argued that RD and SR are inefficient, but could perhaps be improved by good heuristics for choosing next rules. Anyway, hard to evaluate what counts as "fast".
- **Recognize global ambiguity?** CKY builds all parses, so definitely yes. RD/SR can return multiple parses if run past the first one. But notice a distinction...

Serial versus parallel parsing

- Depth-first parsers are inherently **serial**: one structure processed at a time. So, recognizing ambiguity implies backtracking/re-parsing, and one structure will always be recognized *first*.
- Breadth-first parsers are idealized as **parallel**: multiple structures processed simultaneously. If truly parallel, ambiguous structures identified simultaneously.

Human parsing: serial or parallel?

- On the face of it, full parallel parsing seems implausible:
  - Finds/notices all the global ambiguities.
  - Doesn't get stuck in garden paths.
- Serial parsing provides a possible explanation for garden paths (backtracking).
- But there are parallel options too:
  - **limited parallelism**: pursue a fixed (small) number of structures at once, may still require occasional backtracking.
  - **ranked parallelism**: (possibly in combination with above). Possible structures ranked by preference; garden path if low-ranked structure turns out to be correct.

What about incrementality?

First, what does it mean to be incremental?

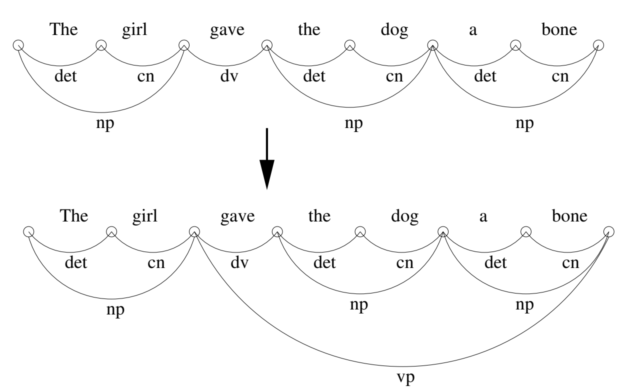
- Each word is integrated into the parse as soon as it is seen/heard.
- Problems with current parse are detected immediately; also possible to make predictions about upcoming words.
- Evidence: eye-tracking of semantic interpretation, garden paths.

Is CKY parsing incremental?

- The chart-filling order we used (short spans first) clearly isn't.
- What about J&M's ordering (fill all cells ending at j, then $j+1$)?
- Consider processing The girl gave the dog a bone:

S	$\rightarrow$	NP VP	Det	$\rightarrow$	the | a
NP	$\rightarrow$	Det CN	CN	$\rightarrow$	girl | dog | bone
VP	$\rightarrow$	TV NP	TV	$\rightarrow$	bit
VP	$\rightarrow$	DV NP NP	DV	$\rightarrow$	gave

- There are still **4** disconnected structures before the rule VP $\rightarrow$ DV NP NP applies, reducing the number to **2** (after which, **1**).



Another problematic example

Consider the garden path sentence the old man the boats.

- Assume a **serial** bottom-up parser (or limited parallel—key is that the correct structure is not considered initially).
- At what point (intuitively) does a human realize the initial analysis is incorrect?
- At what point does the parser realize this?

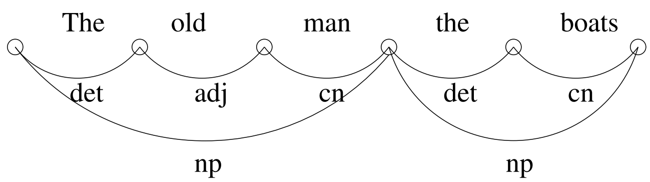
Another problematic example

Grammar for processing the old man the boats:

S	$\rightarrow$	NP VP	Det	$\rightarrow$	the | a
NP	$\rightarrow$	Det CN	Adj	$\rightarrow$	old
NP	$\rightarrow$	Det Adj CN	CN	$\rightarrow$	man | boats | old
VP	$\rightarrow$	TV NP	TV	$\rightarrow$	man | like
VP	$\rightarrow$	DV NP NP	DV	$\rightarrow$	gave

Garden path is too late!

The bottom-up parser doesn't realize its mistake until it reaches the end of the sentence, and cannot create a full parse:



But humans recognize a problem at the second the: they have an **expectation** about what should come next, and it is violated.

Summary so far

- RD parsing cannot model humans because of problems with (eg.) left recursion.
- SR parsing cannot model humans because it doesn't recognize garden paths immediately.
- CKY parsing cannot model humans because it is too parallel, or (if limited), also doesn't recognize garden paths immediately.
- So, where does that leave us?

Left Corner Parsing

Left corner parsing is more cognitively plausible: each word is immediately integrated into a single evolving structure which makes predictions about what will come next.

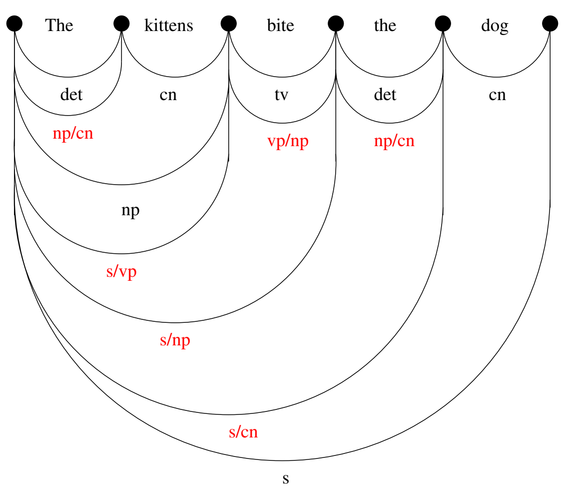
- Mixed directionality: constrained by input (like bottom-up) but also making predictions (like top-down).
- Chart contains **active edges**: incomplete constituents representing predictions.
- Ex: `NP/CN` is an incomplete constituent that will become a complete `NP` if a `CN` is seen next (cf. categorial grammar).

Rules of Left Corner Parsing

1. **Projection**: For a completed edge Y and a grammar rule $X \rightarrow Y\ Z$, add an active edge X/Z, where Y and X/Z span the same part of the string.
2. **Completion**: For an active edge X/Y and a completed edge Y that are adjacent, add a completed edge X that spans the width of both.
3. **Composition**: For two adjacent active edges X/Y and Y/Z, add an active edge X/Z that spans the width of both.

Rule 3 is not necessary for LC parsing, but is necessary for a fully incremental version (i.e., to ensure a single connected structure).

Example of a left corner chart:



Rules of Left Corner Parsing

If dealing with non-binary grammar:

1. **Projection**: For a completed edge Y and a grammar rule $X \rightarrow Y\ \alpha$, add an active edge X/α, where Y and X/α span the same part of the string.
2. **Partial Completion**: For an active edge $X/Y\,\alpha$ and a completed edge Y that are adjacent, add an active edge X/α that spans the width of both.
3. **Completion**: For an active edge X/Y and a completed edge Y that are adjacent, add a completed edge X that spans the width of both.
4. **Composition**: For two adjacent active edges X/Y and Y/Z, add an active edge X/Z that spans the width of both.

Example LC parse for garden path sentence

To try on your own with the grammar provided earlier: the old man the boats

Confirm that the parser realizes a problem where it should!

Summary

- Left-corner parsing achieves full incrementality by using chart entries to represent partial/predictive syntactic structure.
- Looks promising for modelling ambiguity resolution and garden paths.
- But still haven't explained why some parses are preferred or some locally ambiguous sentences (but not others) cause garden paths.
- Other open research issues:
  - Developing fully incremental parsers for wide range of grammar formalisms (some easier than others).
  - How/when does semantics fit in?