Lecture 5: INTRODUCTION TO PARSING

S G Pulman

stephen.pulman@comlab.ox.ac.uk

- Parsing presupposes a grammar and lexicon
- A parsing algorithm uses these to assign (usually all) possible syntactic structures to a given input
- we can ask whether a parsing problem is decidable; whether an algorithm is sound and complete; what its complexity is.
- we usually also need it to be efficient
- Top down, bottom up, depth first, breadth first
- Earley's algorithm; Shift reduce parsing; the CYK algorithm

TOP DOWN PARSING FOR CONTEXT-FREE GRAMMARS

CFG = $\langle$StartSymbol,Terminals,NonTerminals,Rules$\rangle$ where rules are of the form NonTerminal $\rightarrow$ (NonTerminal|Terminal)*

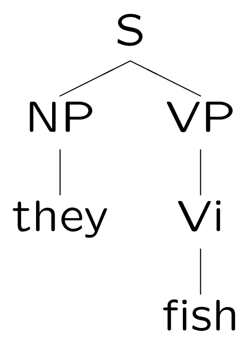
S	$\rightarrow$	NP VP
NP	$\rightarrow$	they | fish
VP	$\rightarrow$	Aux VP
VP	$\rightarrow$	Vi
VP	$\rightarrow$	Vt NP
Aux	$\rightarrow$	can
Vi	$\rightarrow$	fish
Vt	$\rightarrow$	can

Begin with the start symbol
Loop:
Look at the daughters of the rules the current symbol figures in as the mother.
Does the first daughter match the next word in the input?
If it does, do the same for the remaining daughters.
If not, loop with each of the daughters in turn as the current symbol

EXAMPLE

NB the • symbol ‘dot’ precedes the daughter we are currently working on, and the level of indentation indicates which rule has invoked the current action.

Rule	Input
S → •NP VP	they fish
NP → •they	they fish
NP → they•	fish
S → NP •VP	fish
VP → •Vi	fish
Vi → •fish	fish
Vi → fish•	
VP → Vi•	
S → NP VP•	



NON-DETERMINISM

We operate on a stack of 'items': (perhaps partial) constituents, and a list of remaining words.

(i) if there is a word after a dot in the most recent tree that matches the next word in the input, make a new item like the current one except that the dot follows the word, and the next word is removed from the input.

$<$ ... [NP • they], [they can fish] $> \Rightarrow <$ ... [NP they •], [can fish] $>$

(ii) if the most recent tree has the dot at the end of its daughters, integrate it with the tree to its left, if there is one.

$<$[S • NP VP], [NP they•], [can fish]$> \Rightarrow <$[S [NP they] • VP], [can fish]$>$

(iii) if there is a rule whose left hand side matches the category following the dot in the most recent tree, make a new item like the old one with the addition of a new tree derived from the rules, with the dot preceding its daughters.

$<$[S • NP VP], [they can fish]$> \Rightarrow <$[S • NP VP], [NP • they], [they can fish]$>$

Sample parse of ‘they fish’:

0. <[• S], [they fish]>
1. < [S • NP VP], [they fish]>
2. < [S • NP VP], [NP • they], [they fish]>
3. < [S • NP VP], [NP • fish], [they fish]>
4. < [S • NP VP], [NP they•], [fish]>
5. < [S [NP they] • VP], [fish]>
6. < [S [NP they] • VP], [VP • Vi], [fish]>
7. < [S [NP they] • VP], [VP • Vt NP], [fish]>

etc.

Notice that both 2 and 3 were produced from 1, but that 3 was discarded because none of the actions applied to it.

The order in which new items are processed can be varied in order to give depth first or breadth first search behaviour.

The top down algorithm may go into a loop if we encounter ‘left recursion’, e.g. of the category NP:

NP $\rightarrow$ NP ’s N - possessive NPs like [[John] ’s sister]

Now from an item like:

$<$ S $\rightarrow \bullet$ NP VP ...$>$

we would produce one like:

$<$ NP $\rightarrow \bullet$ NP ’s N ....$>$

and this would in turn produce (among others) another identical item:

$<$ NP $\rightarrow \bullet$ NP ’s N ....$>$

and so on.

Two solutions: either rewrite the grammar to eliminate left recursion, or add an extra step to the algorithm to check for repeated items.

NB also `A -> ...... -> A`

Earley's algorithm is a more sophisticated version of the top down procedure we have just introduced informally. It produces sets of items, where each item is augmented with a pointer to the position where it started, by iterating the following three operations. The initial set is S(0) containing just the item:
$<\Sigma \rightarrow \bullet\ \alpha, 0>$, where Σ is the start symbol of the grammar. (Other greek letters represent (possibly empty) sequences of symbols.)

Predict: For every item in S(k) of the form:
$<X \rightarrow \alpha \bullet Y\ \beta, j>$, (where j is the position where the item started), add $<Y \rightarrow\bullet\ \gamma, k>$ to S(k) for every production with Y on the left-hand side, if not already present.

Scan: If a is the next symbol in the input stream, for every item in S(k) of the form $<X \rightarrow \alpha \bullet a\ \beta, j>$, add
$<X \rightarrow \alpha\ a \bullet \beta, j>$ to S(k+1).

Complete: For every item in S(k) of the form $<X \rightarrow \gamma \bullet, j>$, find items in S(j) of the form $<Y \rightarrow \alpha \bullet X\ \beta, i>$ and add $<Y \rightarrow \alpha\ X \bullet \beta, i>$ to S(k).

BOTTOM UP PARSING

We proceed by matching words to the right hand sides of rules, and then matching the resulting symbols, or sequences of symbols, to the right hand sides of rules until we have an S covering the whole sentence.

```
Structure                           Input
so far                              remaining

                                    [they fish]
[NP  they]                          [fish]
[NP  they][Vi  fish]                []
[NP  they][Vp [Vi  fish]]           []
[S [NP  they][Vp [Vi  fish]]]       []
```

NON-DETERMINISTIC VERSION operates on items, as before:

(i) (SHIFT) if there is a rule whose right hand side matches the next word in the input, create a new item with a new tree made from that rule on the top of the constituent stack, and the remainder of the input.

```
<[], [they fish]> ==> <[NP they], [fish] >
```

(ii) (REDUCE) if there is a rule with n daughters matching the n most recent trees on the list, in order, create a new item which combines those daughters into an instance of the mother:

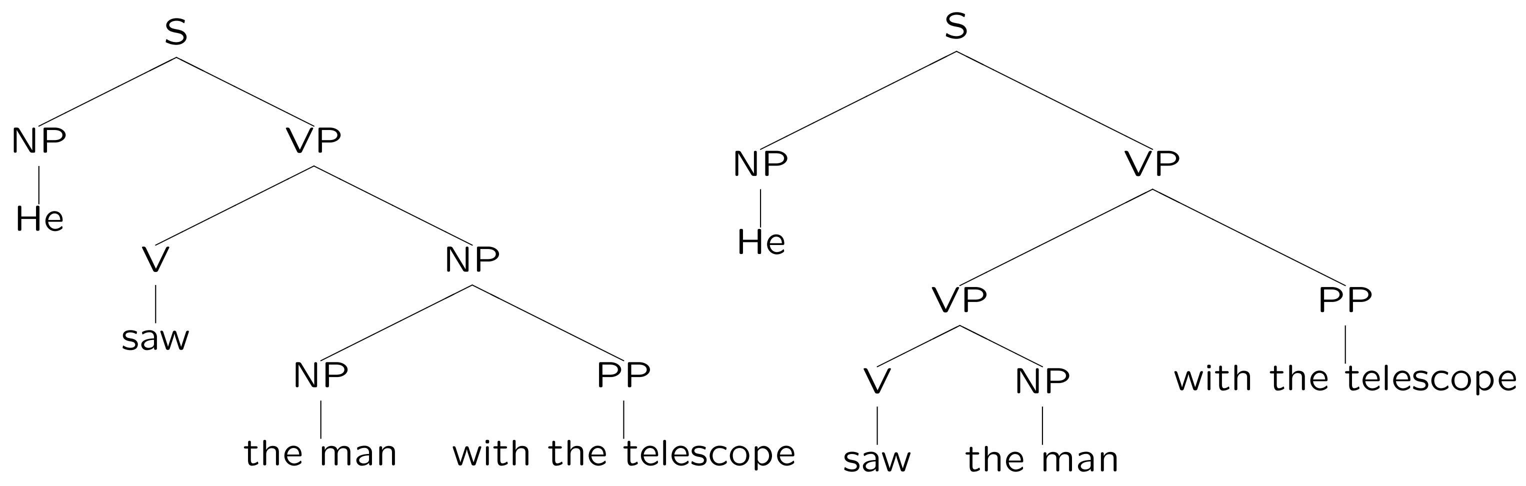
```
< [NP they], [VP [Vi fish]], []> ==> < [S [NP they][VP [Vi fish]]], []>
```

A successful parse is represented by an item with no more input and a single S rooted tree on the stack.

```
1. []                               [they fish]
2. [NP they],                       [fish]
3. [NP they], [NP fish]             []
4. [NP they], [Vi fish]             []
5. [NP they], [VP [Vi fish]]        []
6. [S [NP they] [VP [Vi fish]]]     []
```

Nothing can happen to item 3 and it is discarded.

Naive non-determinism leads to a lot of repeated work:



If we are parsing top down or bottom up, there will at some point be items of the form:

< .... [VP [VP saw the man] • PP], [with the telescope]>
< .... [NP [NP the man] • PP], [with the telescope]>

and both of these will lead to exactly the same sequence of actions to parse the PP ‘with the telescope’. This is very wasteful - consider what would happen in a sentence like:

> *He saw the man with a telescope in a park near the museum with the exhibition about dinosaurs*

The Cocke-Younger-Kasami parsing algorithm for context free grammars given below presupposes that the grammar is in 'Chomsky normal form', i.e. that rules are all of the form A $\rightarrow$ BC, or A $\rightarrow$ T, where A, B, and C are single non-terminal symbols, and T is a single terminal symbol. Any context-free grammar can be transformed to Chomsky Normal Form by a simple transformation on rules not already in that form: see Aho and Ullman p 152 for details.

Example grammar:

```
S    --> NP VP
NP   --> they
NP   --> fish
VP   --> V NP
VP   --> Aux VP
Aux  --> can
V    --> can
V    --> fish
VP   --> V
```

Note that the last two rules need to be merged to '`VP --> fish`' for the grammar to be in Chomsky normal form.

The CYK parsing algorithm

The algorithm builds, for an n-word sentence, an n-by-n table t in which the cells $t_{i,j}$ contain the labels of constituents spanning the sequence of words from position i to j.

Initialise:

```
for i = 1 to n do
   set $t_{i,1}$ to $\{A \mid A \rightarrow T\}$
```

Parse:

```
for j = 2 to n do
   for i = 1 to (n-j)+1 do
     for k = 1 to j-1 do
       set $t_{i,j}$ to $t_{i,j} \bigcup \{A \mid A \rightarrow BC, B \text{ in } t_{i,k}, C \text{ in } t_{i+k,j-k}\}$
```

Trace of CYK algorithm on simple example:

```
S   --> NP VP
NP  --> they
NP  --> fish
VP  --> V NP
VP  --> Aux VP
Aux --> can
V   --> can
VP  --> fish

Input: They can fish

Initialising...
t(1,1,[NP])
t(2,1,[Aux,V])
t(3,1,[NP,VP])
```

```
Parsing...

J=2
 I=1
  K=1,I+K=2,J-K=1
  t(I,K)=t(1,1)
  t(I+K,J-K)=t(2,1)
  t(1,2,[]) - no rules match
 I=2
  K=1,I+K=3,J-K=1
  t(I,K)=t(2,1)
  t(I+K,J-K)=t(3,1)
  Aux is in t(2,1), VP is in t(3,1),
   and VP --> Aux VP, so adding VP to t(2,2)
  V is in t(2,1), NP is in t(3,1),
   and VP --> V NP, so adding VP to t(2,2)
  t(2,2,[VP])
```

```
J=3
 I=1
  K=1,I+K=2,J-K=2
   t(I,K)=t(1,1)
   t(I+K,J-K)=t(2,2)
   NP is in t(1,1), VP is in t(2,2),
   and S --> NP VP, so adding S to t(1,3)
   t(1,3,[S])
  K=2,I+K=3,J-K=1
   t(I,K)=t(1,2)
   t(I+K,J-K)=t(3,1)
   t(1,3,[S])
```

REFERENCES

J. Allen, Chapter 3.

A. V. Aho and J. D. Ullman 1972 The Theory of Parsing, Translation and Compiling, Vol 1 Prentice Hall.

D. Jurafsky and J Martin, Chapter 10 (First edition); Chapter 13 (Second edition)

H. S. Thompson and G. D. Ritchie 1984
Implementing Natural Language Parsers, in T. O'Shea and M. Eisenstadt, eds., Artificial Intelligence: Tools, Techniques, and Applications, N.Y., Harper and Row, 245-300.

T. Winograd 1983 Language as a Cognitive Process, Addison Wesley, especially chapter 3.

J. Earley, "An efficient context-free parsing algorithm", Communications of the Association for Computing Machinery, 13:2:94-102, 1970. `http://portal.acm.org/citation.cfm?doid=362007.362035`

CYK demo:
`http://homepages.uni-tuebingen.de/student/martin.lazarov/demos/cky.html`