CS 181: Natural Language Processing

Lecture 10: Parsing

Kim Bruce
Pomona College
Spring 2008

Disclaimer: Slide contents borrowed from many sources on web!

Example CFG

- T = {this, that, a, the, man, book, flight, meal, include, read, does}
- N = {S, NP, NOM, VP, Det, Noun, Verb, Aux}
- S - start
- R =

S → NP VP	VP → Verb
S → Aux NP VP	VP → Verb NP
S → VP	Det → that | this | a | the
NP → Det NOM	Noun → book | flight | meal | man
NOM → Noun	Verb → book | include | read
NOM → Noun NOM	Aux → does

Why Parsing?

- Machine translation:
  - L1 ⇒ PT1 ⇒ PT2 ⇒ L2
- Speech synthesis from parsing:
  - The government plans to raise income tax.
  - The government plans to raise income tax the imagination.
  - Speech recognition:
    - Put the file in the folder.
    - Put the file and the folder.

Why Parsing?

- Grammar Checking
- Indexing for information retrieval
- Information extraction
  - Subject vs. object

Human Language Parsing

Human Language Processing

- Seven principles from Kimball, 1973, Cognition 2:15-47
  1. Top-down: parsing in natural language proceeds according to a top-down algorithm
  2. Right association: Sentences organize into right-branching structures (less complex)
  3. New nodes: A new node is signalled by a function word (preps, det, conjunctions, complementizers, auxs, wh-words)

Human Language Processing

4. Two sentences: Max of two sentences can be parsed in parallel
   - *That that Joe left bothered Susan surprised Max*
5. Closure: A phrase is closed as soon as possible (unless the next node is a constituent of the phrase)
   - *They knew that the girl was in the closet*
   - *They knew the girl was in the closet*
6. Fixed structure: Costly to reorganize the constituent after a phrase has been closed
   - *Garden path sentences*

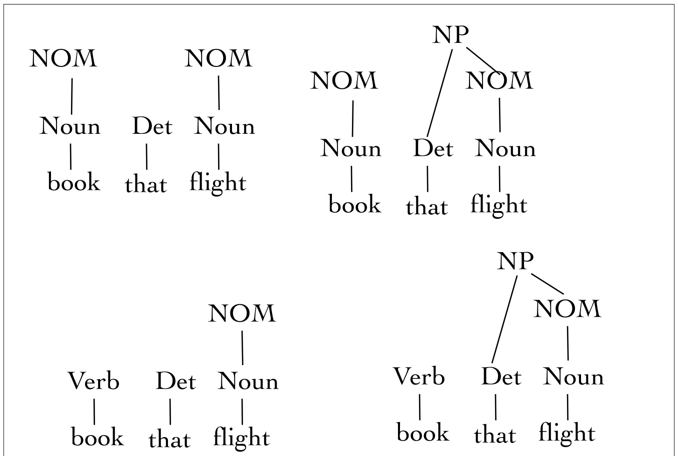
Human Language Processing

7. Processing: When a phrase is closed, it is pushed down into a syntactic processing stage and cleared from short-term memory
   - *Tom saw that the cow jumped over the moon.*

Bottom-Up Parsing

Bottom-Up Parsing

- Data-driven: Start w/ string. Rewrite by replacing RHS by LHS of rules until get S.
- May have several RHS matches.
- Usually presented as shift-reduce parse
  - YACC



Shift-Reduce

sentence → NounPhrase VerbPhrase
NounPhrase → Art Noun
VerbPhrase → Verb | Adverb Verb
Art → the | a | ...
Verb → jumps | sings | ...
Noun → dog | cat | ...

Parse: *The dog jumps*

Draw trees as parse!

```
Stack                    Input Sequence
()                       (the dog jumps)
(the)                    (dog jumps)      SHIFT word onto stack
(Art)                    (dog jumps)      REDUCE using grammar rule
(Art dog)                (jumps)          SHIFT..
(Art Noun)               (jumps)          REDUCE..
(NounPhrase)             (jumps)          REDUCE
(NounPhrase jumps)       ()               SHIFT
(NounPhrase Verb)        ()               REDUCE
(NounPhrase VerbPhrase)  ()               REDUCE
(Sentence)               ()               SUCCESS
```

Bottom-Up Parsing

- Do we shift or reduce?
- If reduce, which rule do we use?
- With prog. langs, build table to always tell you what to do -- deterministic.
- Programming languages designed to be unambiguous. We don't have that luxury!
- ε-rules can be applied anywhere!
- May need to backtrack!

Top-Down vs. Bottom-Up

- Top-down may explore paths that can never result in desired string
  - In prog. langs, can make sure that doesn't happen.
- Bottom up may build subtrees that can not be part of trees rooted at S.
- Both may have to repeat work when backtracking!

Keys to Success

- Watch out for bad grammars
  - left-recursive for top-down (VP → VP PP)
- Try to avoid redoing work when backtracking
- Grammar transformations help
  - ... but linguists will hate you!

CKY Parsing

Dynamic Programming: CKY Parser

- Given CFG in CNF and an input string, produce the collection of all valid parse trees.
- Think recursively: what about last step in building a parse tree for subsequence of input.
  - Suppose root is labeled A.
  - If non-trivial, top production is A → B C
  - Thus, string w produced by A can be written w_B w_C where $B \rightarrow^* w_B$, $C \rightarrow^* w_B$
  - Need to search to see where to divide w.

Dynamic Programming

- Number gaps between words:
  - $_0$ Time $_1$ flies $_2$ like $_3$ an $_4$ arrow $_5$.
- Create n × n upper-triangular table
  - rows: 0 to n-1.
  - cols: 1 to n.
  - cell[i,j] contains non-terminals that could head a subtree generating words between i and j
  - E.g., cell[3,5] contains NP

EXAMPLE GRAMMAR IN CNF

NP → time	S → NP VP
Vst → time	S → Vst NP
NP → flies	S → S PP
VP → flies	VP → V NP
P → like	VP → VP PP
V → like	NP → Det N
Det → an	NP → NP PP
N → arrow	NP → NP NP
	PP → P NP

ALGORITHM

```
function CKY_Parse(words, grammar)
  n ← length(words)
  for w ← 1 to n do
    table[w-1,w] ← {A | A → words[w] ∈ grammar}
    for start ← 0 to n-w do  # start is row
      end ← start + w         # end is column
      for mid ← start+1 to end-1
        for every X in table[start,mid]
          for every Y in table[mid,end]
            for all B s.t B → X Y ∈ grammar
              add B to table[start,end]
```

CREATING A TABLE

- Enter the part of speech for $word_i$ in cell[i-1,i]

0	Time $_1$	flies $_2$	like $_3$	an $_4$	arrow $_5$
0	NP, Vst				
1		NP, VP			
2			P, V		
3				Det	
4					N

FILLING IN TABLE

0	Time $_1$	flies $_2$	like $_3$	an $_4$	arrow $_5$
0	NP, Vst	*NP*			
1		NP, VP			
2			P, V		
3				Det	
4					N

$NP \rightarrow NP_1\ NP$

FILLING IN TABLE

0	Time $_1$	flies $_2$	like $_3$	an $_4$	arrow $_5$
0	NP, Vst	NP, S^2			
1		NP, VP			
2			P, V		
3				Det	
4					N

$S \rightarrow NP_1\ VP,\ S \rightarrow Vst_1\ NP$

FILLING IN TABLE

0	Time $_1$	flies $_2$	like $_3$	an $_4$	arrow $_5$
0	NP, Vst	NP, S^2			
1		NP, VP	-		
2			P, V	-	
3				Det	*NP*
4					N

$NP \rightarrow Det_4\ N$

Filling In Table

0	Time 1	flies 2	like 3	an 4	arrow 5
0	NP, Vst	NP, S^2	-		↘
1		NP, VP	-	-	↘
2			P, V	-	*VP, PP*
3				Det	NP
4					N

$VP \rightarrow V_3\ NP$, $PP \rightarrow P_3\ NP$

Filling In Table

0	Time 1	flies 2	like 3	an 4	arrow 5
0	NP, Vst	NP, S^2	-	-	↘
1		NP, VP	-	-	*S, NP, VP*
2			P, V	-	VP, PP
3				Det	NP
4					N

$S \rightarrow NP_2\ VP$, $NP \rightarrow NP_2\ PP$, $VP \rightarrow VP_2\ PP$

Filling In Table

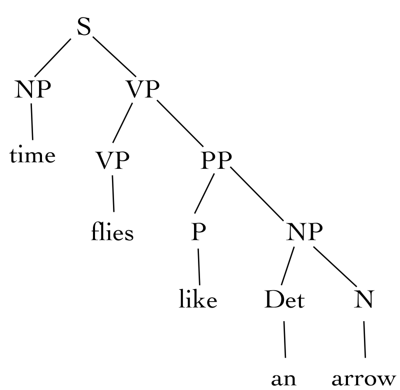
0	Time 1	flies 2	like 3	an 4	arrow 5
0	NP, Vst	NP, S^2	-	-	*S^5, NP^2*
1		NP, VP	-	-	S, NP, VP
2			P, V	-	VP, PP
3				Det	NP
4					N

$S \rightarrow NP_1\ VP$, $NP \rightarrow NP_1\ NP$, $S \rightarrow Vst_1\ NP$,
$NP \rightarrow NP_2\ PP$, $S \rightarrow NP_2\ VP$, $S \rightarrow S_2\ PP$

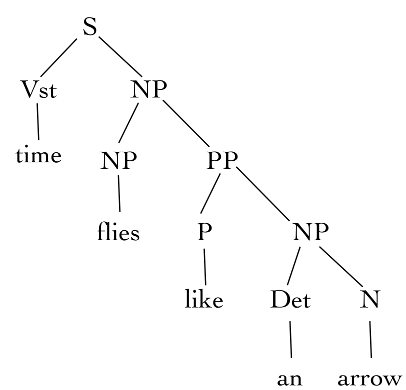
Backpointer to Parse Trees

- Each entry in table corresponds to a parse tree
- Reconstruct using backpointers or could actually associate tree with each entry (sharing subtrees, for efficiency)

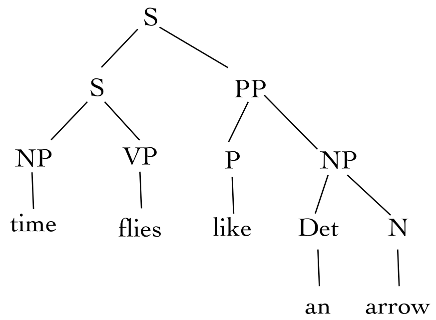
Correct Parse



Incorrect Parse



Incorrect Parse



Exercise

$_0$	She $_1$	eats $_2$	fish $_3$	with $_4$	chopsticks $_5$
0	NP				
1					
2					
3					
4					

NP → she	V → eats	NP → NP PP
NP → fish	V → fish	VP → V NP
NP → fork	P → with	VP → VP PP
NP → chopsticks	S → NP VP	PP → P NP

Earley Algorithm

Earley Algorithm

- Top-down
- Does not require CNF, handles left-recursion.
- Proceeds left-to-right filling in a chart
- States contain 3 pieces of info:
  - Grammar rule
  - Progress made in recognizing it
  - Position of subtree in input string

Parse Table

- As before, columns correspond to gaps
- Entry in column n of the form
  - A → u.v, k
  - Means predicting that we'll use rule A → u v, and so far have verified u in input matches section of input [k,n]
- Ex: $_0$ Book $_1$ that $_2$ flight $_3$
  - NP → Det.Nom,1 in column 2 means have recognized "that" (word[1,2]) is Det and hope to show Nom occurs later

Earley Algorithm

Add **ROOT → . S** to column 0.

For each j from 0 to n:

For each dotted rule in column j,
(including those added as we go!)

look at what's after the dot:

- If it's a word w, SCAN:
  - If w matches the input word between j and j+1, advance the dot and add the new rule to column j+1
- If it's a non-terminal X, PREDICT:
  - Add all rules for X to the bottom of column j, with the dot at the start: e.g. **X → . Y Z**
- If there's nothing after the dot, ATTACH:
  - We've finished some constituent, A, that started in column i<j. So for each rule in column j that has A after the dot: Advance the dot and add the result to the bottom of column j.

Return true if last column has **ROOT → S .**

Idea of Algorithm

- Process all hypotheses in order
- May add new hypotheses (or try to add old)
- Process according to what after dot
  - if word, scan and see if matches
  - if non-terminal, predict ways to match
    - if want, can be smart and peek ahead to reduce possibilities
  - if at end, have complete constituent and attach to those that need it.

Example

S → NP VP	VP → Verb
S → Aux NP VP	VP → Verb NP
S → VP	Det → that | this | a | the
NP → Det NOM	Noun → book | flight | meal | man
NOM → Noun	Verb → book | include | read
NOM → Noun NOM	Aux → does

Book that flight!

Earley Example

chart[0]

book

ROOT→.S, 0	
S → .NP VP, 0	
S → .Aux NP VP, 0	✘
S → .VP, 0	
NP → . Det Nom, 0	✘
VP → . Verb, 0	
VP → . Verb NP, 0	

3 predictions for S

✘ *because book is not Aux or Det*

Any Questions?