CKY and Earley Algorithms
Chapter 13

Lecture #8

October 2012

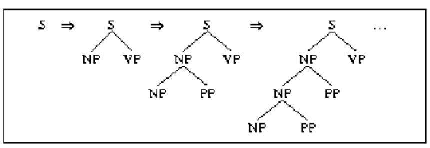
Review

- Top-Down vs. Bottom-Up Parsers
  - Both generate too many useless trees
  - Combine the two to avoid over-generation: Top-Down Parsing with Bottom-Up look-ahead
- Left-corner table provides more efficient look-ahead
  - Pre-compute all POS that can serve as the leftmost POS in the derivations of each non-terminal category
- More problems remain..

Left Recursion

- Depth-first search will never terminate if grammar is left recursive (e.g. NP --> NP PP)

$$(A \xrightarrow{*} \alpha AB, \alpha \xrightarrow{*} \varepsilon)$$



Left-Recursion

- What happens in the following situation
  - S -> NP VP
  - S -> Aux NP VP
  - NP -> NP PP
  - NP -> Det Nominal
  - …
  - With the sentence starting with
    - Did the flight…

Rule Ordering

- S -> Aux NP VP
- S -> NP VP
- NP -> Det Nominal
- NP -> NP PP

- The key for the NP is that you want the recursive option after any base case. Duhh.

Rule Ordering

- S -> Aux NP VP
- S -> NP VP
- NP -> Det Nominal
- NP -> NP PP

- What happens with
  - Book that flight

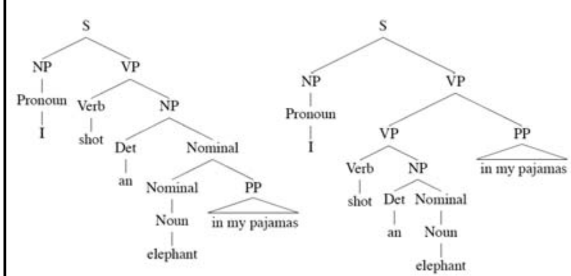
- Solutions:
  - Rewrite the grammar (automatically?) to a weakly equivalent one which is not left-recursive
    - e.g. The man {on the hill with the telescope…}
    - NP → NP PP
    - NP → Nom PP
    - NP → Nom
    - …becomes…
    - NP → Nom NP'
    - NP' → PP NP'
    - NP' → e
    - This may make rules unnatural

- Harder to eliminate non-immediate left recursion
  - NP --> Nom PP
  - Nom --> NP
- Fix depth of search explicitly
- Rule ordering: non-recursive rules first
  - NP --> Det Nom
  - NP --> NP PP

Structural ambiguity:

- Multiple legal structures
  - Attachment (e.g., I saw a man on a hill with a telescope)
  - Coordination (e.g., younger cats and dogs)
  - NP bracketing (e.g., Spanish language teachers)

Ambiguity

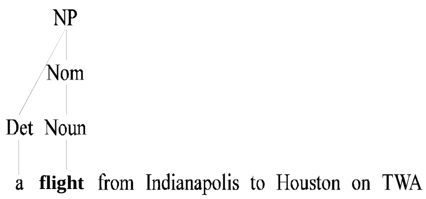


- Solution?
  - Return all possible parses and disambiguate using "other methods"

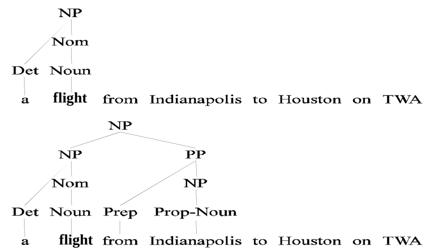
Avoiding Repeated Work

- Parsing is hard, and slow. It's wasteful to redo stuff over and over and over.
- Consider an attempt to top-down parse the following as an NP

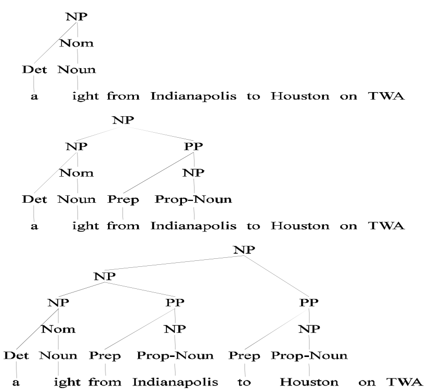
A flight from India to Houston on TWA



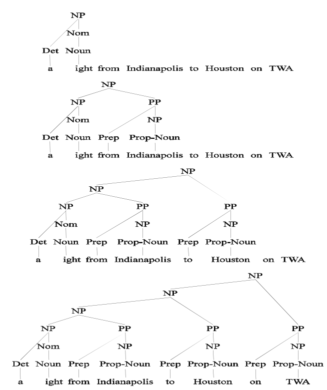
13



14



15



16

Dynamic Programming

- We need a method that fills a table with partial results that
  - Does not do (avoidable) repeated work
  - Does not fall prey to left-recursion
  - Solves an exponential problem in polynomial time (sort of)

17

Dynamic Programming

- Create table of solutions to sub-problems (e.g. subtrees) as parse proceeds
- Look up subtrees for each constituent rather than re-parsing
- Since all parses implicitly stored, all available for later disambiguation
- Examples: Cocke-Younger-Kasami (CYK) (1960), Graham-Harrison-Ruzzo (GHR) (1980) and Earley (1970) algorithms

18

Dynamic Programming

- DP search methods fill tables with partial results and thereby
  - Avoid doing avoidable repeated work
  - Solve exponential problems in polynomial time (well, no not really)
  - Efficiently store ambiguous structures with shared sub-parts.
- We'll cover two approaches that roughly correspond to top-down and bottom-up approaches.
  - CKY
  - Earley

CKY Parsing

- First we'll limit our grammar to epsilon-free, binary rules (more later)
- Consider the rule *A* → *BC*
  - If there is an A somewhere in the input then there must be a B followed by a C in the input.
  - If the A spans from i to j in the input then there must be some k st. i<k<j
    - Ie. The B splits from the C someplace.

Problem

- What if your grammar isn't binary?
  - As in the case of the TreeBank grammar?
- Convert it to binary… any arbitrary CFG can be rewritten into Chomsky-Normal Form automatically.
- What does this mean?
  - The resulting grammar accepts (and rejects) the same set of strings as the original grammar.
  - But the resulting derivations (trees) are different.

Problem

- More specifically, we want our rules to be of the form

A → B C

Or

A → *w*

That is, rules can expand to either 2 non-terminals or to a single terminal.

Binarization Intuition

- Eliminate chains of unit productions.
- Introduce new intermediate non-terminals into the grammar that distribute rules with length > 2 over several rules.
  - So… *S → A B C turns into*

S → X C and

X → A B

Where X is a symbol that doesn't occur anywhere else in the the grammar.

Sample L1 Grammar

Grammar	Lexicon
S → NP VP	*Det → that | this | a*
S → Aux NP VP	*Noun → book | flight | meal | money*
S → VP	*Verb → book | include | prefer*
NP → Pronoun	*Pronoun → I | she | me*
NP → Proper-Noun	*Proper-Noun → Houston | NWA*
NP → Det Nominal	*Aux → does*
Nominal → Noun	*Preposition → from | to | on | near | through*
Nominal → Nominal Noun	
Nominal → Nominal PP	
VP → Verb	
VP → Verb NP	
VP → Verb NP PP	
VP → Verb PP	
VP → VP PP	
PP → Preposition NP	

CNF Conversion

$\mathscr{L}_1$ Grammar	$\mathscr{L}_1$ in CNF
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow TWA \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

CKY

- So let's build a table so that an A spanning from i to j in the input is placed in cell [i,j] in the table.
- So a non-terminal spanning an entire string will sit in cell [0, n]
  - Hopefully an *S*
- If we build the table bottom-up, we'll know that the parts of the A must go from i to k and from k to j, for some k.

CKY

- Meaning that for a rule like A → B C we should look for a B in [i,k] and a C in [k,j].
- In other words, if we think there might be an A spanning i,j in the input… AND

  A → B C is a rule in the grammar THEN
- There must be a B in [i,k] and a C in [k,j] for some i<k<j

CKY

- So to fill the table loop over the cell[i,j] values in some systematic way
  - What constraint should we put on that systematic search?
  - For each cell, loop over the appropriate k values to search for things to add.

CKY Algorithm

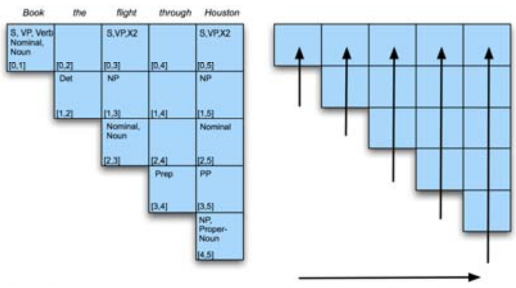
```
function CKY-PARSE(words, grammar) returns table

  for j ← from 1 to LENGTH(words) do
      table[j−1, j] ← {A | A → words[j] ∈ grammar}
      for i ← from j−2 downto 0 do
          for k ← i+1 to j−1 do
              table[i,j] ← table[i,j] ∪
                            {A | A → BC ∈ grammar,
                                 B ∈ table[i,k],
                                 C ∈ table[k,j]}
```

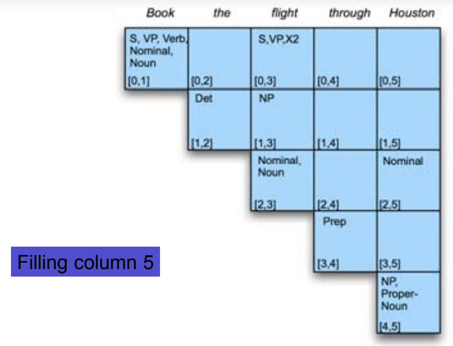

Note

- We arranged the loops to fill the table a column at a time, from left to right, bottom to top.
  - This assures us that whenever we're filling a cell, the parts needed to fill it are already in the table (to the left and below)
  - It's somewhat natural in that it processes the input a left to right a word at a time
    - Known as online

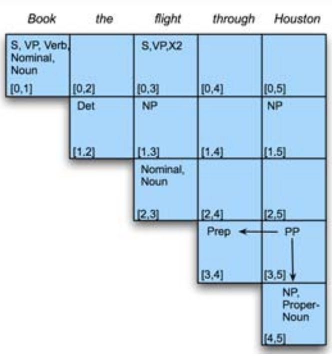
Example



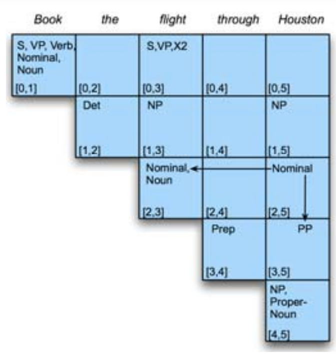
Example



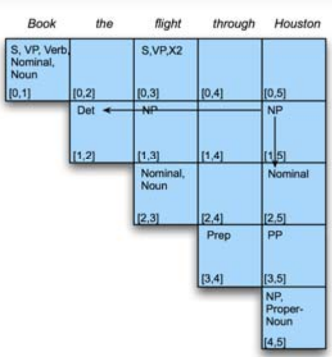
Example



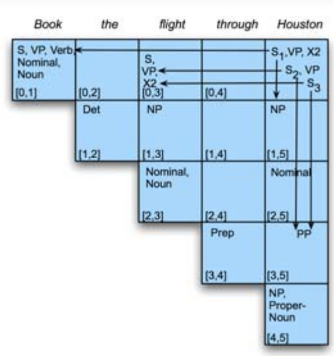
Example



Example



Example



CKY Notes

- Since it's bottom up, CKY populates the table with a lot of phantom constituents.
  - Segments that by themselves are constituents but cannot really occur in the context in which they are being suggested.
  - To avoid this we can switch to a top-down control strategy
  - Or we can add some kind of filtering that blocks constituents where they can not happen in a final analysis.

Earley's Algorithm

- Uses dynamic programming to do parallel top-down search in (worst case) $O(N^3)$ time
- First, L2R pass fills out a chart with N+1 states (N: the number of words in the input)
  - Think of chart entries as sitting between words in the input string keeping track of states of the parse at these positions
  - For each word position, chart contains set of states representing all partial parse trees generated to date. E.g. chart[0] contains all partial parse trees generated at the beginning of the sentence

Earley Parsing

- Fills a table in a single sweep over the input words
  - Table is length N+1; N is number of words
  - Table entries represent
    - Completed constituents and their locations
    - In-progress constituents
    - Predicted constituents

States

- The table-entries are called states and are represented with dotted-rules.

S -> · VP	A VP is predicted
NP -> Det · Nominal	An NP is in progress
VP -> V NP ·	A VP has been found

States/Locations

- It would be nice to know where these things are in the input so…[x,y] tells us where the state begins (x) and where the dot lies (y) wrt the input

S -> · VP [0,0]		A VP is predicted at the start of the sentence
NP -> Det · Nominal		[1,2] An NP is in progress; the Det goes from 1 to 2
VP -> V NP ·	[0,3]	A VP has been found starting at 0 and ending at 3

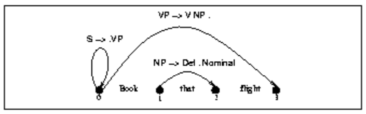
$_0$ Book $_1$ that $_2$ flight $_3$

S --> • VP, [0,0]
- First 0 means S constituent begins at the start of the input
- Second 0 means the dot is there too
- So, this is a top-down prediction

NP --> Det • Nom, [1,2]
- the NP begins at position 1
- the dot is at position 2
- so, Det has been successfully parsed
- Nom predicted next

VP --> V NP •, [0,3]

- Successful VP parse of entire input



Successful Parse

- Final answer found by looking at last entry in chart
- If entry resembles S --> α • [0,N] then input parsed successfully
- But note that chart will also contain a record of all possible parses of input string, given the grammar -- not just the successful one(s)

Earley

- As with most dynamic programming approaches, the answer is found by looking in the table in the right place.
- In this case, there should be an S state in the final column that spans from 0 to n+1 and is complete.
- If that's the case you're done.
  - S – α · [0,n+1]

Earley

- So sweep through the table from 0 to n+1…
  - New predicted states are created by starting top-down from S. In each case, use rules in the grammar to expand a state when what is being looked for is a non-terminal symbol. A new prediction is made for every rule that has that non-terminal as its left-hand side.
  - New incomplete states are created by advancing existing states as new constituents are discovered
  - New complete states are created in the same way.

Earley

- More specifically…
  1. Predict all the states you can upfront
  2. Read a word
     1. Extend states based on matches
     2. Add new predictions
     3. Go to 2
  3. When you are out of words, look in the chart at N+1 to see if you have a winner

Parsing Procedure for the Earley Algorithm

- Move through each set of states in order, applying one of three operators to each state:
  - predictor: add predictions to the chart
  - scanner: read input and add corresponding state to chart
  - completer: move dot to right when new constituent found
- Results (new states) added to current or next set of states in chart
- No backtracking and no states removed: keep complete history of parse

Core Earley Code

```
function EARLEY-PARSE(words, grammar) returns chart

 ENQUEUE((γ → • S, [0,0]), chart[0])
 for i ← from 0 to LENGTH(words) do
  for each state in chart[i] do
   if INCOMPLETE?(state) and
          NEXT-CAT(state) is not a part of speech then
      PREDICTOR(state)
   elseif INCOMPLETE?(state) and
          NEXT-CAT(state) is a part of speech then
      SCANNER(state)
   else
      COMPLETER(state)
  end
 end
 return(chart)
```


Earley Code

```
procedure PREDICTOR((A → α • B β, [i, j]))
   for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
      ENQUEUE((B → • γ, [j, j]), chart[j])
   end

procedure SCANNER((A → α • B β, [i, j]))
   if B ⊂ PARTS-OF-SPEECH(word[j]) then
      ENQUEUE((B → word[j], [j, j+1]), chart[j+1])

procedure COMPLETER((B → γ •, [j,k]))
   for each (A → α • B β, [i, j]) in chart[j] do
      ENQUEUE((A → α B • β, [i,k]), chart[k])
   end
```


Predictor

- Intuition: new states represent top-down expectations
- Applied when non part-of-speech non-terminals are to the right of a dot
  S --> • VP [0,0]
- Adds new states to ***current*** chart
  - One new state for each expansion of the non-terminal in the grammar
    VP --> • V [0,0]
    VP --> • V NP [0,0]

Scanner

- New states for predicted part of speech.
- Applicable when part of speech is to the right of a dot
  VP --> • V NP [0,0] 'Book…'
- Looks at current word in input
- If match, adds state(s) to ***next*** chart
  VP --> V • NP [0,1]

Completer

- Intuition: parser has discovered a constituent, so must find and advance all states that were waiting for this
- Applied when dot has reached right end of rule
  NP --> Det Nom • [1,3]
- Find all states w/dot at 1 and expecting an NP
  VP --> V • NP [0,1]
- Adds new (completed) state(s) to ***current*** chart
  VP --> V NP • [0,3]

Core Earley Code

```
function EARLEY-PARSE(words, grammar) returns chart

 ENQUEUE((γ → • S, [0,0]), chart[0])
 for i ← from 0 to LENGTH(words) do
  for each state in chart[i] do
   if INCOMPLETE?(state) and
          NEXT-CAT(state) is not a part of speech then
      PREDICTOR(state)
   elseif INCOMPLETE?(state) and
          NEXT-CAT(state) is a part of speech then
      SCANNER(state)
   else
      COMPLETER(state)
  end
 end
 return(chart)
```

Earley Code

```
procedure PREDICTOR((A → α • B β, [i, j]))
    for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
        ENQUEUE((B → • γ, [j, j]), chart[j])
    end

procedure SCANNER((A → α • B β, [i, j]))
    if B ⊂ PARTS-OF-SPEECH(word[j]) then
        ENQUEUE((B → word[j], [j, j+1]), chart[j+1])

procedure COMPLETER((B → γ •, [j, k]))
    for each (A → α • B β, [i, j]) in chart[j] do
        ENQUEUE((A → α B • β, [i, k]), chart[k])
    end
```


$_0$ Book $_1$ that $_2$ flight $_3$ (Chart [0])

- Seed chart with top-down predictions for S from grammar

γ → • S	[0,0]	Dummy start state
S → • NP VP	[0,0]	Predictor
S → • Aux NP VP	[0,0]	Predictor
S → • VP	[0,0]	Predictor
NP → • Det Nom	[0,0]	Predictor
NP → • PropN	[0,0]	Predictor
VP → • V	[0,0]	Predictor
VP → • V NP	[0,0]	Predictor

CFG for Fragment of English

S → NP VP	Det → that | this | a
S → Aux NP VP	N → book | flight | meal | money
S → VP	V → book | include | prefer
NP → Det Nom	Aux → does
Nom → N	
Nom → N Nom	Prep →from | to | on
NP →PropN	PropN → Houston | TWA
VP → V	Nom → Nom PP
VP → V NP	PP → Prep NP

- When dummy start state is processed, it's passed to Predictor, which produces states representing every possible expansion of S, and adds these and every expansion of the left corners of these trees to bottom of Chart[0]
- When VP --> • V, [0,0] is reached, Scanner called, which consults first word of input, Book, and adds first state to Chart[1], V --> Book •, [0,1]
- Note: When VP --> • V NP, [0,0] is reached in Chart[0], Scanner expands the rule yielding VP --> V . NP, [0,1] but does not put V --> Book •, [0,1] in again.

Example

Chart[0]

γ ■ ▪ S	[0,0]	Dummy start state
S ■ ▪ NP VP	[0,0]	Predictor
NP ■ ▪ Det NOMINAL	[0,0]	Predictor
NP ■ ▪ Proper-Noun	[0,0]	Predictor
S ■ ▪ Aux NP VP	[0,0]	Predictor
S ■ ▪ VP	[0,0]	Predictor
VP ■ ▪ Verb	[0,0]	Predictor
VP ■ ▪ Verb NP	[0,0]	Predictor

Chart[1]

Verb ■ book ▪	[0,1]	Scanner
VP ■ Verb▪	[0,1]	Completer
S ■ VP▪	[0,1]	Completer
VP ■ Verb ▪ NP	[0,1]	Completer
NP ■ ▪ Det NOMINAL	[1,1]	Predictor
NP ■ ▪ Proper-Noun	[1,1]	Predictor

Chart[1]

V→ book •	[0,1]	Scanner
VP → V •	[0,1]	Completer
VP → V • NP	[0,1]	Completer
S → VP •	[0,1]	Completer
NP → • Det Nom	[1,1]	Predictor
NP → • PropN	[1,1]	Predictor

V--> book • passed to Completer, which finds 2 states in Chart[0,0] whose left corner is V and adds them to Chart[0,1], moving dots to right

- When VP → V • is itself processed by the Completer, S → VP • is added to Chart[1] since VP is a left corner of S
- Last 2 rules in Chart[1] are added by Predictor when VP → V • NP is processed
- And so on….

61

Example

Chart[1]		
Verb ■ *book* ▪	[0,1]	Scanner
VP ■ *Verb*▪	[0,1]	Completer
S ■ *VP*▪	[0,1]	Completer
VP ■ *Verb* ▪ *NP*	[0,1]	Completer
NP ■ ▪ *Det NOMINAL*	[1,1]	Predictor
NP ■ ▪ *Proper-Noun*	[1,1]	Predictor

Chart[2]		
Det ■ *that*▪	[1,2]	Scanner
NP ■ *Det*▪*NOMINAL*	[1,2]	Completer
NOMINAL ■ ▪ *Noun*	[2,2]	Predictor
NOMINAL ■ ▪ *Noun NOMINAL*	[2,2]	Predictor

62

Example

Chart[2]		
Det ■ *that*▪	[1,2]	Scanner
NP ■ *Det*▪*NOMINAL*	[1,2]	Completer
NOMINAL ■ ▪ *Noun*	[2,2]	Predictor
NOMINAL ■ ▪ *Noun NOMINAL*	[2,2]	Predictor

Chart[3]		
Noun ■ *ight*▪	[2,3]	Scanner
NOMINAL ■ *Noun*▪	[2,3]	Completer
NOMINAL ■ *Noun*▪ *NOMINAL*	[2,3]	Completer
NP ■ *Det NOMINAL* ▪	[1,3]	Completer
VP ■ *Verb NP* ▪	[0,3]	Completer
S ■ *VP*▪	[0,3]	Completer
NOMINAL ■ ▪ *Noun*	[3,3]	Predictor
NOMINAL ■ ▪ *Noun NOMINAL*	[3,3]	Predictor

63

What is it?

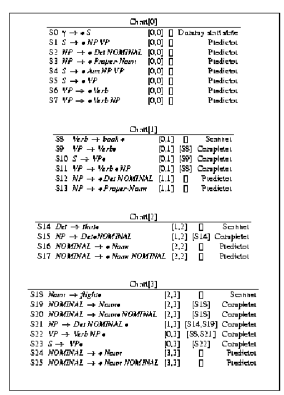
- What kind of parser did we just describe (trick question).
  - Earley parser… yes
  - Not a parser – a recognizer
    - The presence of an S state with the right attributes in the right place indicates a successful recognition.
    - But no parse tree… no parser

64

How do we retrieve the parses at the end?

- Augment the Completer to add ptr to prior states it advances as a field in the current state
  - I.e. what state did we advance here?
  - Read the ptrs back from the final state
- Do we NEED the pointers?

65



Chart[0]			
S0 γ → • S	[0,0]	[]	Dummy start state
S1 S → • NP VP	[0,0]	[]	Predictor
S2 NP → • Det NOMINAL	[0,0]	[]	Predictor
S3 NP → • Proper-Noun	[0,0]	[]	Predictor
S4 S → • Aux NP VP	[0,0]	[]	Predictor
S5 S → • VP	[0,0]	[]	Predictor
S6 VP → • Verb	[0,0]	[]	Predictor
S7 VP → • Verb NP	[0,0]	[]	Predictor

Chart[1]			
S8 Verb → book •	[0,1]	[]	Scanner
S9 VP → Verb•	[0,1]	[S8]	Completer
S10 S → VP•	[0,1]	[S9]	Completer
S11 VP → Verb • NP	[0,1]	[S8]	Completer
S12 NP → • Det NOMINAL	[1,1]	[]	Predictor
S13 NP → • Proper-Noun	[1,1]	[]	Predictor

Chart[2]			
S14 Det → that•	[1,2]	[]	Scanner
S15 NP → Det•NOMINAL	[1,2]	[S14]	Completer
S16 NOMINAL → • Noun	[2,2]	[]	Predictor
S17 NOMINAL → • Noun NOMINAL	[2,2]	[]	Predictor

Chart[3]			
S18 Noun → flight•	[2,3]	[]	Scanner
S19 NOMINAL → Noun•	[2,3]	[S18]	Completer
S20 NOMINAL → Noun• NOMINAL	[2,3]	[S18]	Completer
S21 NP → Det NOMINAL •	[1,3]	[S14,S19]	Completer
S22 VP → Verb NP •	[0,3]	[S8,S21]	Completer
S23 S → VP•	[0,3]	[S22]	Completer
S24 NOMINAL → • Noun	[3,3]	[]	Predictor
S25 NOMINAL → • Noun NOMINAL	[3,3]	[]	Predictor

66

Useful Properties

- Error handling
- Alternative control strategies

Error Handling

- What happens when we look at the contents of the last table column and don't find a S --> $\alpha\bullet$ rule?
  - Is it a total loss? No...
  - Chart contains every constituent and combination of constituents possible for the input given the grammar
- Also useful for partial parsing or shallow parsing used in information extraction

Alternative Control Strategies

- Change Earley top-down strategy to bottom-up or ...
- Change to best-first strategy based on the probabilities of constituents
  - Compute and store probabilities of constituents in the chart as you parse
  - Then instead of expanding states in fixed order, allow probabilities to control order of expansion

Summing Up

- Ambiguity, left-recursion, and repeated re-parsing of subtrees present major problems for parsers
- Solutions:
  - Combine top-down predictions with bottom-up look-ahead
  - Use dynamic programming
  - Example: the Earley algorithm
- Next time: Read Ch 15