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

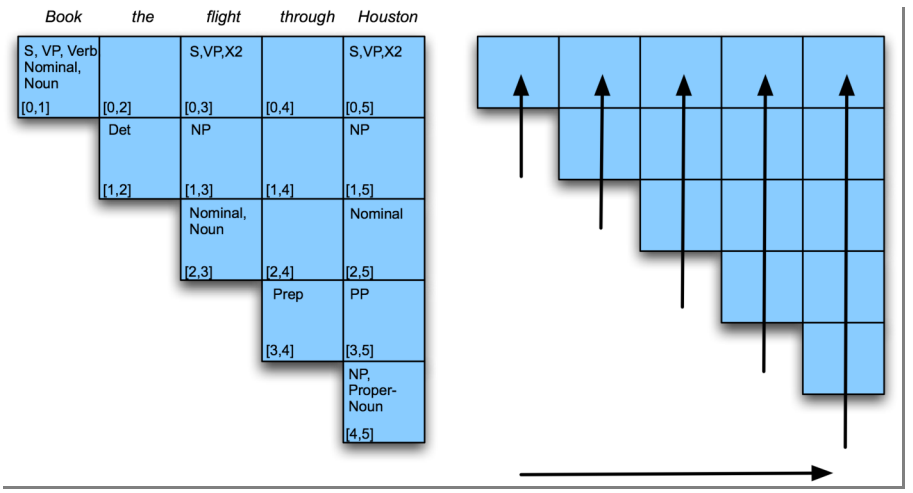
CKY

- 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.

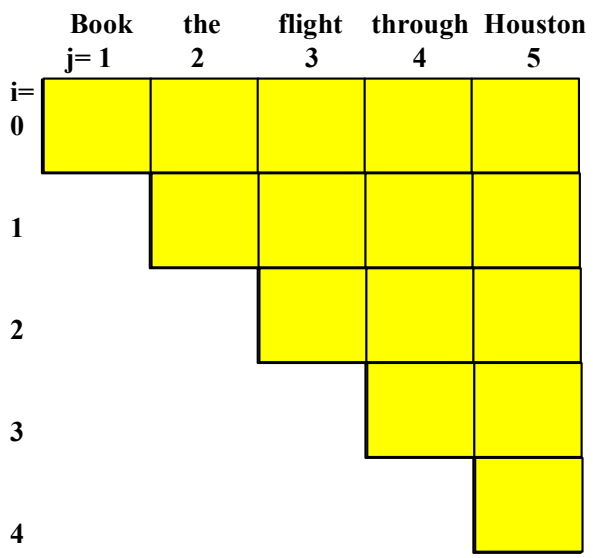
Note

- 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

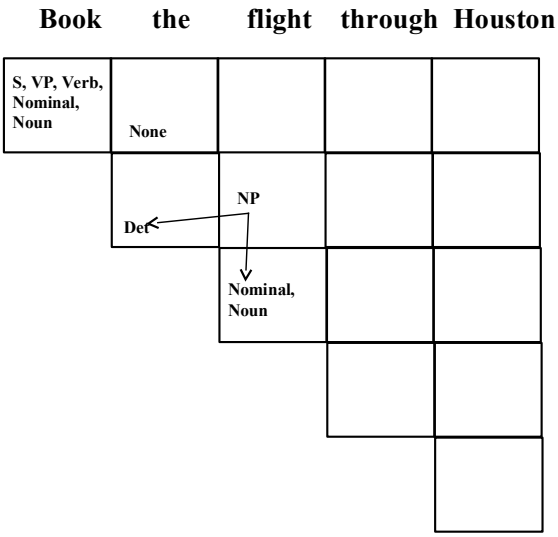


CKY Parser

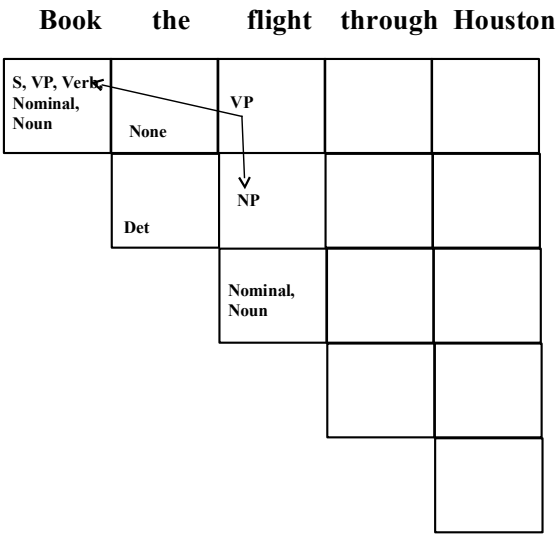


Cell[*i*,*j*] contains all constituents (non-terminals) covering words *i* +1 through *j*

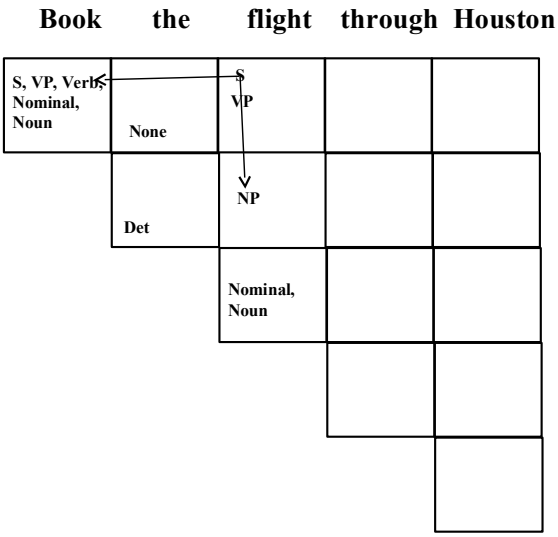
CKY Parser



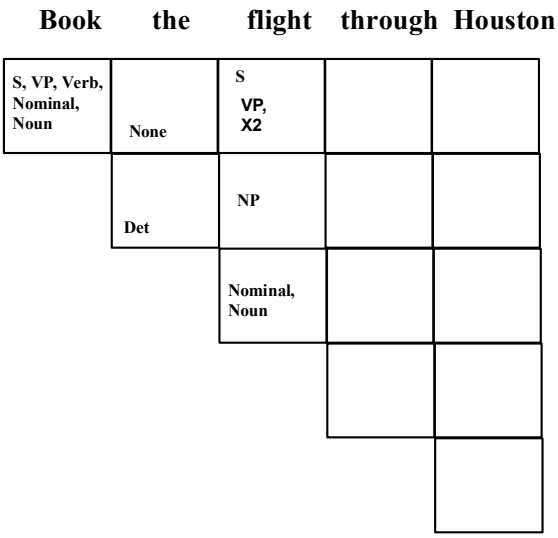
CKY Parser



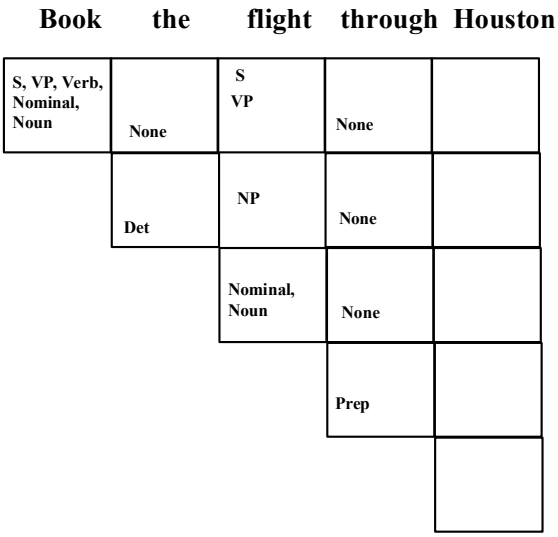
CKY Parser



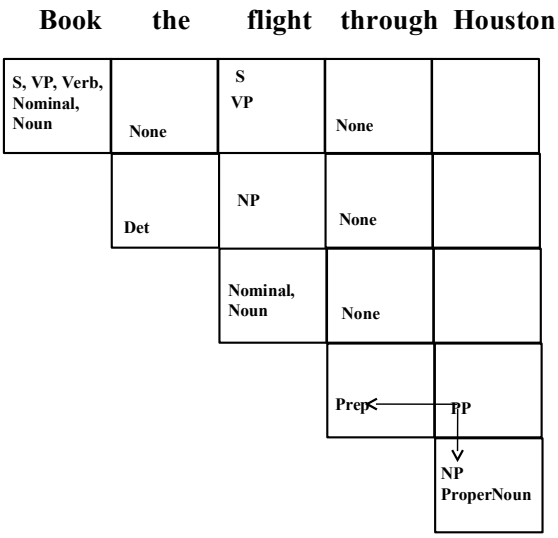
CKY Parser



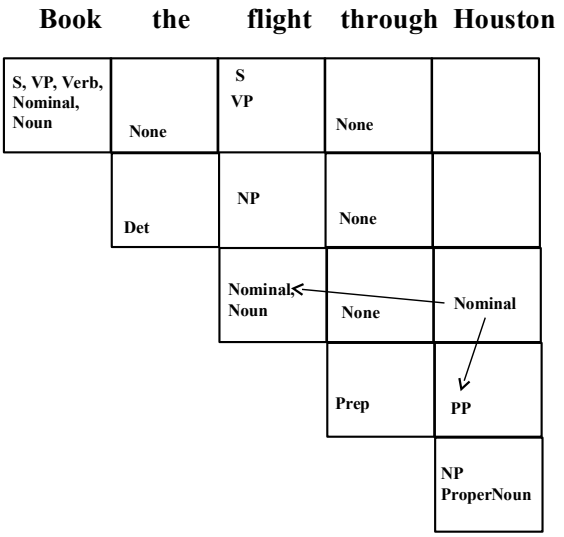
CKY Parser



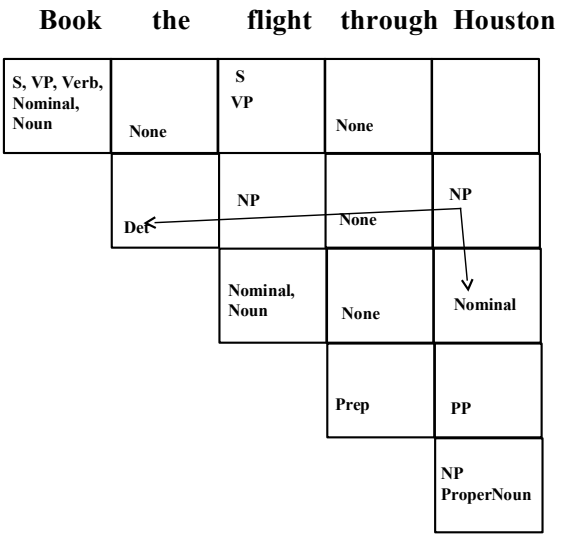
CKY Parser



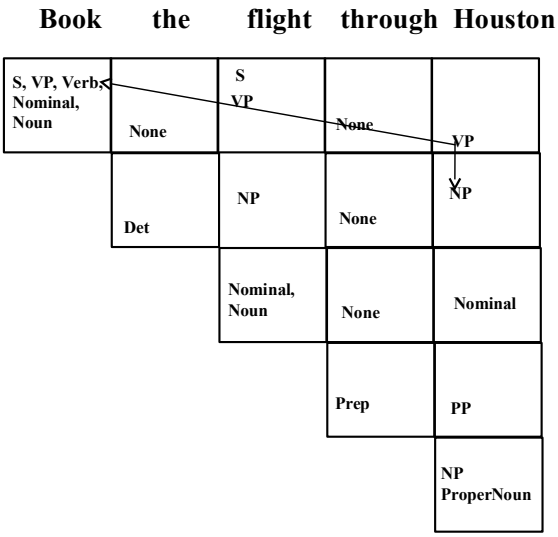
CKY Parser



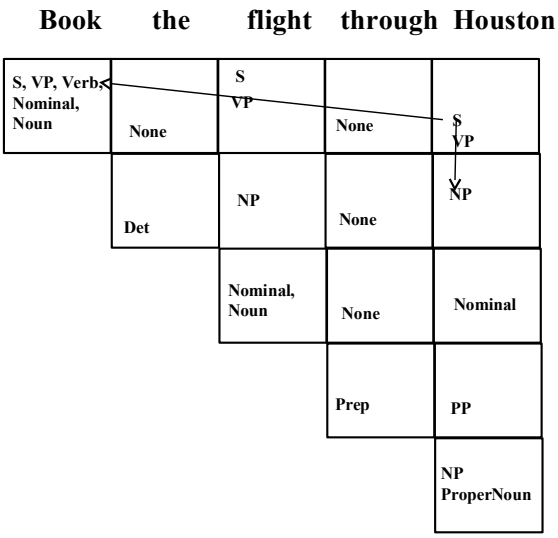
CKY Parser



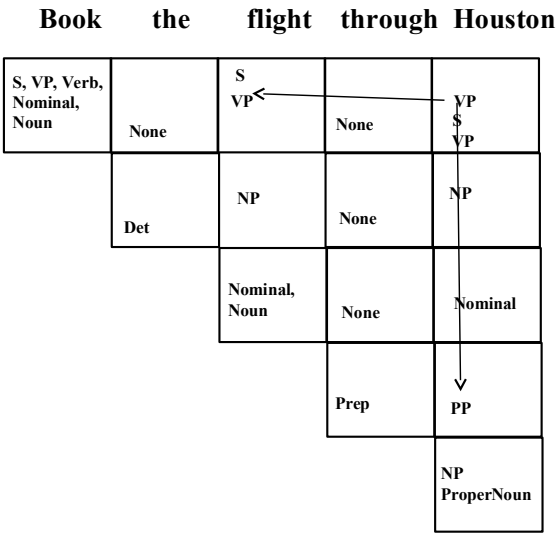
CKY Parser



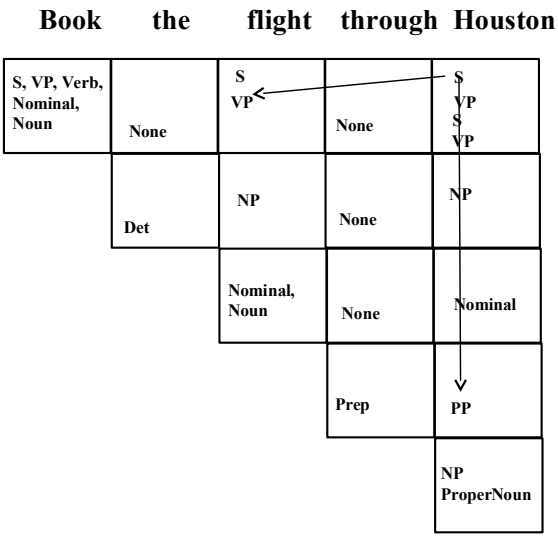
CKY Parser



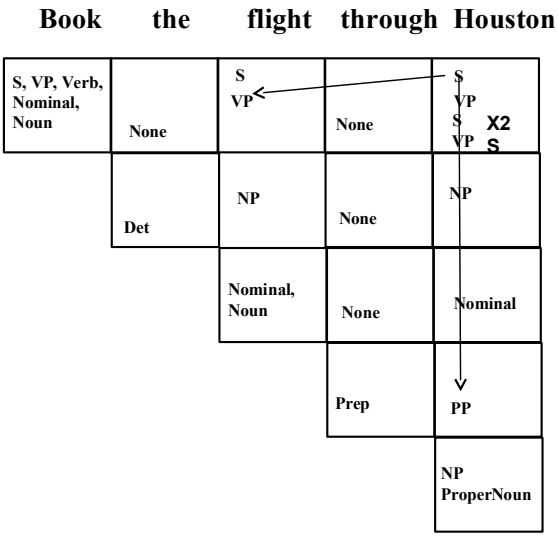
CKY Parser



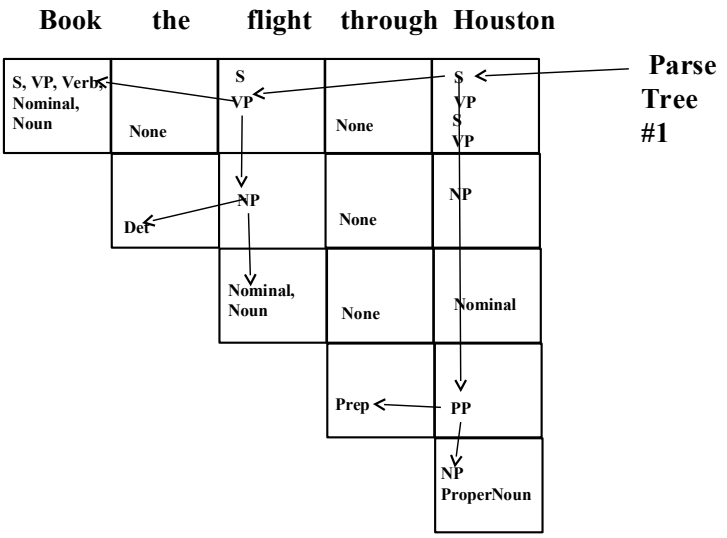
CKY Parser



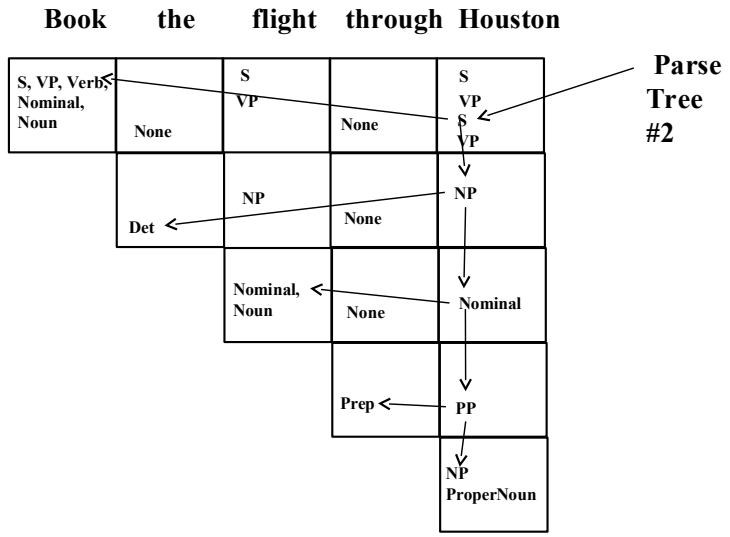
CKY Parser



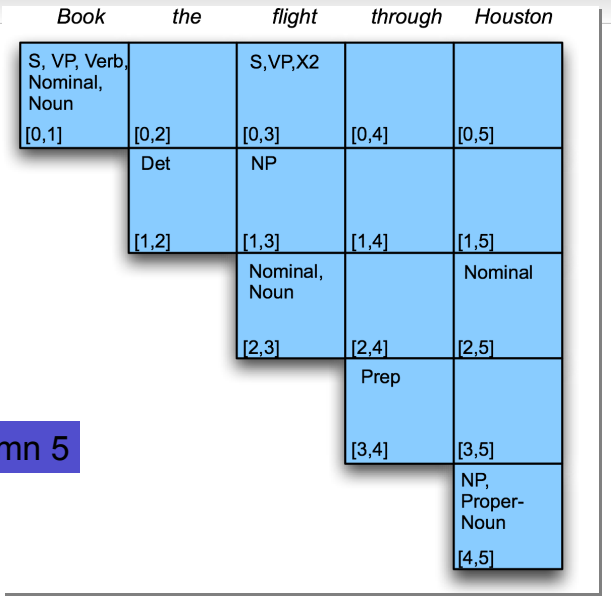
CKY Parser



CKY Parser

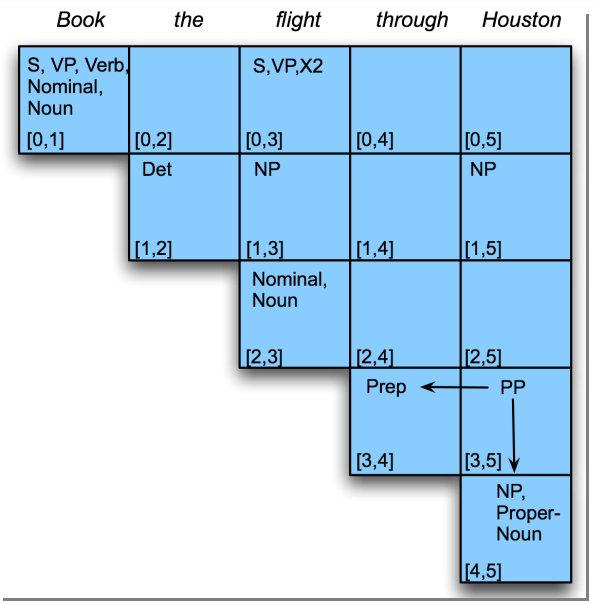


Example

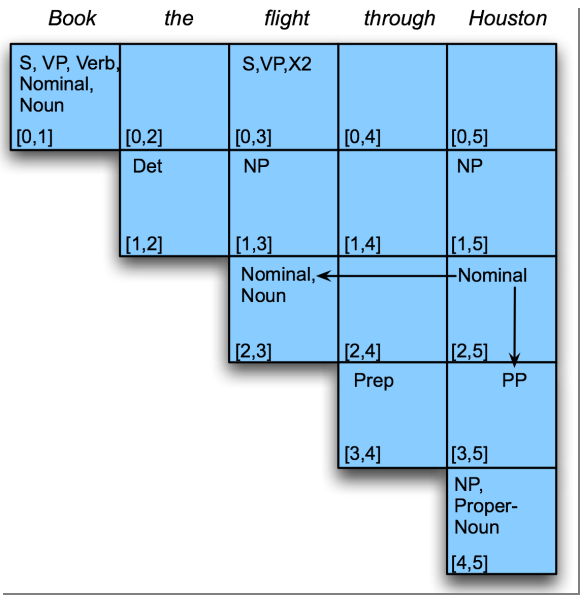


Filling column 5

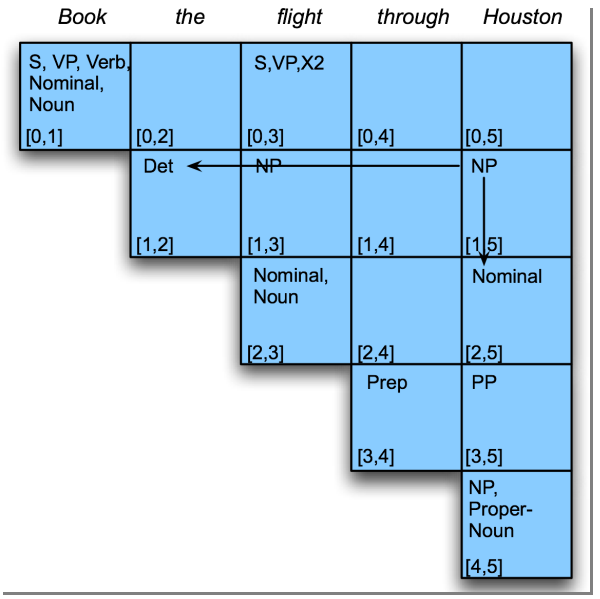
Example



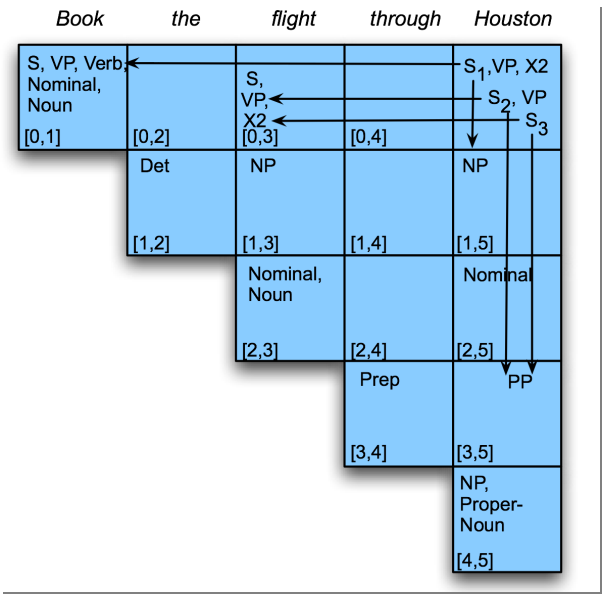
Example



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 Parsing

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

Back to Ambiguity

- Did we solve it?

Ambiguity

- No…
  - Both CKY and Earley will result in multiple S structures for the [0,N] table entry.
  - They both efficiently store the sub-parts that are shared between multiple parses.
  - And they obviously avoid re-deriving those sub-parts.
  - But neither can tell us which one is right.

Ambiguity

- In most cases, humans don't notice incidental ambiguity (lexical or syntactic). It is resolved on the fly and never noticed.
  - I ate the spaghetti with chopsticks
  - I ate the spaghetti with meatballs
- We'll try to model that with probabilities.

Shallow or Partial Parsing

- Sometimes we don't need a complete parse tree
  - Information extraction
  - Question answering
- But we would like more than simple POS sequences

Chunking

- Find major but unembedded constituents like NPs, VPs, AdjPs, PPs
  - Most common task: NP chunking of base NPs
  - [NP I] saw [NP the man] on [NP the hill] with [NP a telescope]
  - No attempt to identify full NPs – no recursion, no post-head words
  - No overlapping constituents
  - E.g., if we add PPs or VPs, they may consist only of their heads, e.g. [PP on]

Approaches: RE Chunking

- Use regexps to identify constituents, e.g.
  - NP → (DT) NN* NN
  - Find longest matching chunk
  - Hand-built rules
  - No recursion but can cascade to approximate true CF parser, aggregating larger and larger constituents

Approaches: Tagging for Chunking

- Require annotated corpus
- Train classifier to classify each element of input in sequence (e.g. IOB Tagging)
  - B (beginning of sequence)
  - I (internal to sequence)
  - O (outside of any sequence)
  - No end-of-chunk coding – it's implicit
  - Easier to detect the beginning than the end

  Book/B_VP that/B_NP flight/I_NP quickly/O

Summary and Limitations

- Sometimes shallow parsing is enough for task
- Performance quite accurate

Distribution of Chunks in CONLL Shared Task

Label	Category	Proportion (%)	Example
NP	Noun Phrase	51	*The most frequently cancelled flight*
VP	Verb Phrase	20	*may not arrive*
PP	Prepositional Phrase	20	*to Houston*
ADVP	Adverbial Phrase	4	*earlier*
SBAR	Subordinate Clause	2	*that*
ADJP	Adjective Phrase	2	*late*

Summing Up

- Parsing as search: what search strategies to use?
  - Top down
  - Bottom up
  - How to combine?
- How to parse as little as possible
  - Dynamic Programming
- Shallow Parsing