EECS 595

Natural Language Processing

Lecture 6: Syntactic Parsing

Instructor: Joyce Chai

Misc.

- Homework 2 is due on October 23

What is Grammar

- A vogue in the 19th century to describe the structures of an area of knowledge

  (e.g., Busby's " A Grammar of Music"

  Field's "A Grammar of Colouring")
- The meaning of grammar:
  - principles or structures as a field of inquiry
  - The grammar of syntax
    - Syntax: setting out together or arrangement; the way words are arranged.

Syntax

- Covered: POS categories
- To be covered:
  - Constituency
  - Grammatical relations
  - Subcategorization and dependencies

What is a constituent?

- The idea: groups of words may behave as a single unit or phrase -> constituent
- Example: noun phrase act like a unit
  - "University of Michigan"
  - "the campus"
  - "well-weathered three-story structure"
- How can we model the constituency?
  - With context-free grammars

Context-Free Grammars

- Chomsky (1956) Backus (1959)
- Set of rules (productions) – expressing the way symbols of the language can be grouped together
  - NP -> Det Nominal
  - NP -> Proper Noun
  - Nominal -> Noun | Noun Nominal
- Lexicon
  - Det -> a
  - Det -> the
  - Noun -> flight

Context Free Grammars

- Capture constituents and ordering
  - Need something else for grammatical relations and dependency relations
- Consists of
  - Set of terminals (words)
  - Set of non-terminal (the constituent of language)
  - Sets of rules of the form A->α, where α is a string of zero or more terminals and non-terminals
- They are equivalent to Backus-Naur form grammars

Formal Definition of CFG

- Set of non-terminal symbols N
- Set of terminals Σ
- Set of productions A -> α

  A $\in$N, α-string $(\Sigma \cup N)^*$
- A designated start symbol

Some NP Rules

- Here are some rules for our noun phrases

$$
\begin{aligned}
NP &\rightarrow Det\ Nominal \\
NP &\rightarrow ProperNoun \\
Nominal &\rightarrow Noun \mid Nominal\ Noun
\end{aligned}
$$

- Together, these describe two kinds of NPs.
  - One that consists of a determiner followed by a nominal
  - And another that says that proper names are NPs.
  - The third rule illustrates two things
    - An explicit disjunction
      - Two kinds of nominals
    - A recursive definition
      - Same non-terminal on the right and left-side of the rule

Example Grammar

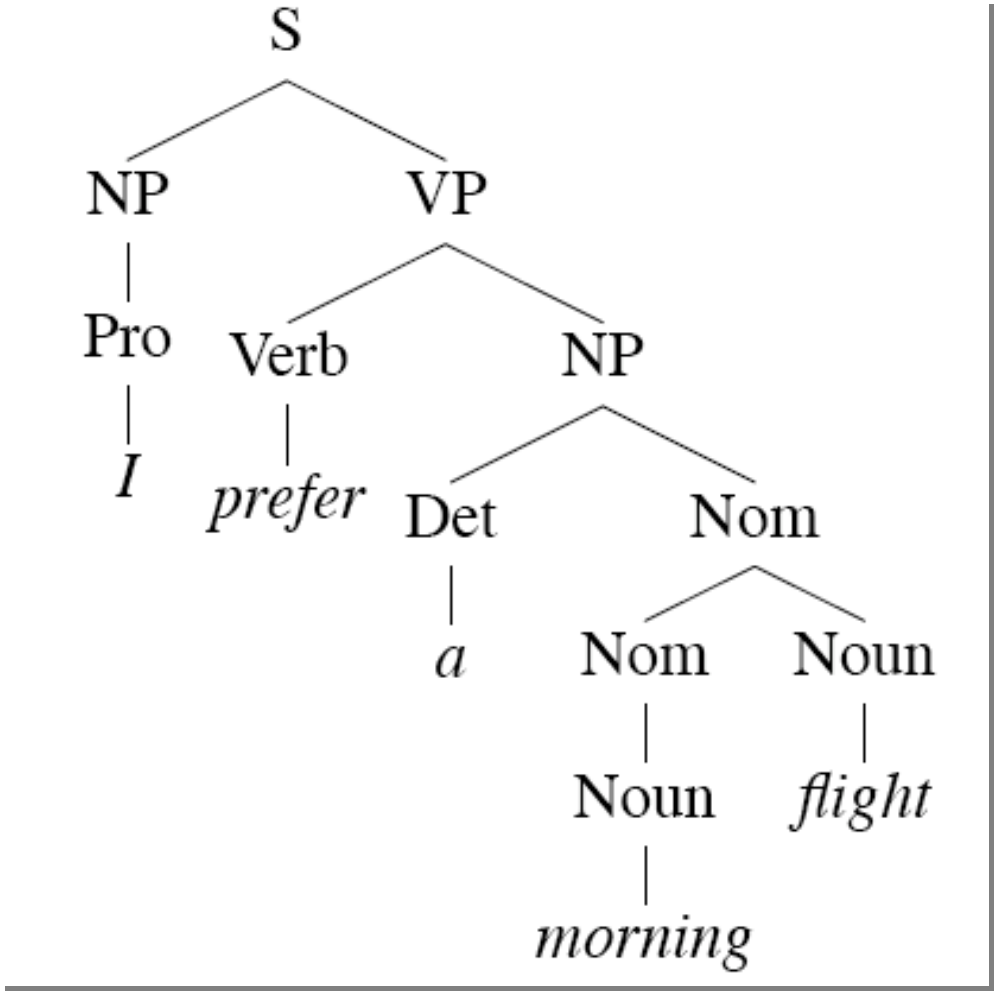
Grammar Rules			Examples
S	→	*NP VP*	I + want a morning flight
NP	→	*Pronoun*	I
	|	*Proper-Noun*	Los Angeles
	|	*Det Nominal*	a + flight
Nominal	→	*Nominal Noun*	morning + flight
	|	*Noun*	flights
VP	→	*Verb*	do
	|	*Verb NP*	want + a flight
	|	*Verb NP PP*	leave + Boston + in the morning
	|	*Verb PP*	leaving + on Thursday
PP	→	*Preposition NP*	from + Los Angeles

Generativity

- You can view these rules as either analysis or synthesis machines
  - Generate strings in the language
  - Reject strings not in the language
  - Impose structures (trees) on strings in the language

Derivations

- A derivation is a sequence of rules applied to a string that *accounts* for that string
  - Covers all the elements in the string
  - Covers only the elements in the string



Key Constituents

- Sentences
- Noun phrases
- Verb phrases
- Prepositional phrases

Sentence Types

- Declaratives

  S –> NP VP (e.g., John left)
- Imperatives

  S -> VP (e.g., Leave!)
- Yes-No Questions

  S -> Aux NP VP (e.g., Did John leave?)
- WH Questions

  S -> Wh-NP VP (e.g., Whose flights serve breakfast?)

  S -> Wh–NP Aux NP VP (e.g., Which flight did you take?)

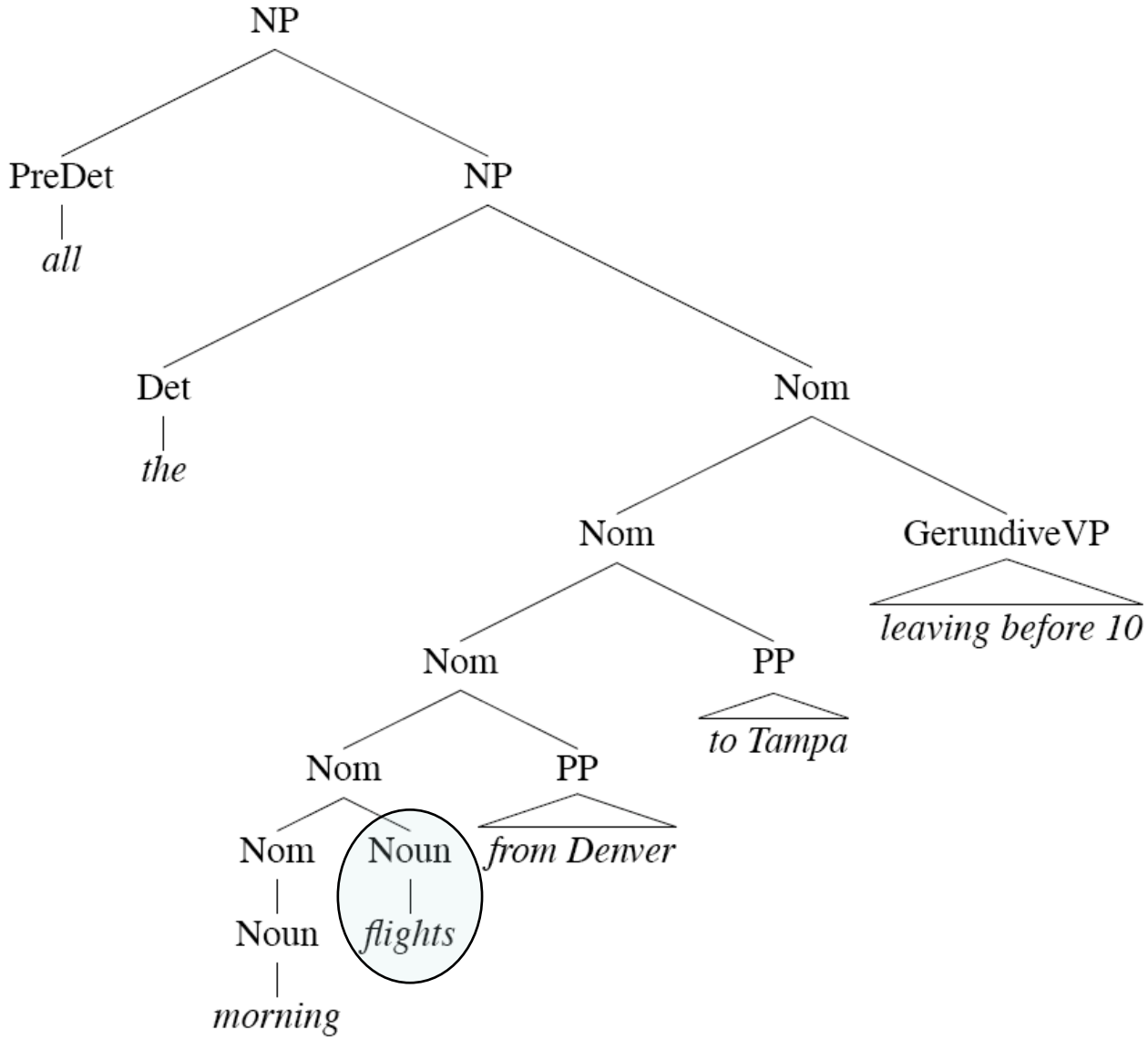
Noun Phrases

- Let's consider the following rule in more detail...

  $NP \rightarrow Det\ Nominal$

- Most of the complexity of English noun phrases is hidden in this rule.
- Consider the derivation for the following example
  - *All the morning flights from Denver to Tampa leaving before 10*

Noun Phrases



NP Structure

- Clearly this NP is really about *flights*. That's the central critical noun in this NP. Let's call that the *head*.
- We can dissect this kind of NP into the part that comes before the head, and the part that comes after it.

Determiners

- Noun phrases can start with determiners...
- Determiners can be
  - Simple lexical items: *the, this, a, an*, etc.
    - A car
  - Or simple possessives
    - John's car
  - Or complex recursive versions of that
    - John's sister's husband's son's car

Nominals

- Contains the head and any pre- and post- modifiers of the head.
  - Pre-
    - Quantifiers, cardinals, ordinals...
      - Three cars
    - Adjectives and Aps
      - large cars
    - Ordering constraints
      - Three large cars
      - ?large three cars

Postmodifiers

- Three kinds
  - Prepositional phrases
    - From Seattle
  - Non-finite clauses (-ing, -ed, and infinitive form)
    - Arriving before noon
  - Relative clauses
    - That serve breakfast
- Same general (recursive) rule to handle these
  - *Nominal → Nominal PP*
  - *Nominal → Nominal GerundVP*
  - *Nominal → Nominal RelClause*

Gerunds

- any flights [arriving after ten p.m]
- *Nominal* → *Nominal GerundVP*
- *GerundVP* → *GerundV NP* | *GerundV PP* | *GerundV* | *GerundV NP PP*
- *GerundV* → *being* | *preferring* | *arriving* …

Infinitives and –ed forms

- the last flight to arrive in Boston
- I need to have dinner served
- which is the aircraft used by this flight?

Postnominal relative clauses

- Restrictive relative clauses:
  - A flight that serves breakfast
  - Flights that leave in the morning
  - The United flight that arrives in San Jose at ten p.m.
- Rules:
  - *Nominal → Nominal RelClause*
  - *RelClause → (who | that) VP*

Combining post-modifiers

- *A flight from Phoenix to Detroit leaving Monday evening*
- *Evening flights from Nashville to Houston that serve dinner*
- *The earliest American Airlines flight that I can get*

Recursive Structure

- Rules where the non-terminal on the left-hand side also appears on the right-hand side.
  - *NP → NP PP (The flight to Boston)*
  - *VP → VP PP (departed Miami at noon)*

Recursive Structure

- Allow us to the following:
  - *Flights to Miami*
  - *Flights to Miami from Boston*
  - *Flights to Miami from Boston in April*
  - *Flights to Miami from Boston in April on Friday*
  - *Flights to Miami from Boston in April on Friday under $300*
  - *….*

Conjunctions

- Any phrasal constituent can be conjoined with a constituent of the same type to form a new constituent of that type
  - *NP → NP and NP*
  - *S → S and S*
- *X -> X and X*

Some Difficulties

- Agreement
- Subcategorization

Agreement

- By ***agreement***, we have in mind constraints that hold among various constituents that take part in a rule or set of rules
- For example, in English, determiners and the head nouns in NPs have to agree in their number.

This flight

Those flights

*This flights

*Those flight

Problem

- Our earlier NP rules are clearly deficient since they don't capture this constraint
  - *NP → Det Nominal*
    - Accepts, and assigns correct structures, to grammatical examples (*this flight*)
    - But its also happy with incorrect examples (*these flight)
  - Such a rule is said to *overgenerate*.
- How to solve the problem?

Agreement

- S → Aux NP VP

- S → 3sgAux 3sgNP VP
- S → Non3sgAux non3sgNP VP

- 3sgAux → does | has | can …
- non3sgAux → do | have | can …

Agreement

- We now need similar rules for NPs, also for number agreement, etc.
  - 3SgNP → (Det) (Card) (Ord) (Quant) (AP) SgNominal
  - Non3SgNP → (Det) (Card) (Ord) (Quant) (AP) PlNominal
  - SgNominal → SgNoun | SgNoun SgNoun
  - etc.
- Combinatorial explosion

Verb Phrases

- English *VP*s consist of a head verb along with 0 or more following constituents which we'll call *arguments*.

$VP \rightarrow Verb$ disappear
$VP \rightarrow Verb\ NP$ prefer a morning flight
$VP \rightarrow Verb\ NP\ PP$ leave Boston in the morning
$VP \rightarrow Verb\ PP$ leaving on Thursday

Subcategorization

- But, even though there are many valid VP rules in English, not all verbs are allowed to participate in all those VP rules.
  - Verbs are compatible with different kinds of complements
- We can subcategorize the verbs in a language according to the sets of VP rules that they participate in.
- Traditional grammar distinguishes between transitive and intransitive.
- Modern grammars may have 100s or such classes.

Subcategorization

- Sneeze: John sneezed
- Find: Please find [a flight to NY]$_{NP}$
- Give: Give [me]$_{NP}$[a cheaper fare]$_{NP}$
- Help: Can you help [me]$_{NP}$[with a flight]$_{PP}$
- Prefer: I prefer [to leave earlier]$_{TO\text{-}VP}$
- Told: I was told [United has a flight]$_{S}$
- …

Subcategorization

- *John sneezed the book
- *I prefer United has a flight
- *Give with a flight

- As with agreement phenomena, we need a way to formally express the constraints

Overgeneration

- Right now, the various rules for VPs *overgenerate*.
  - They permit the presence of strings containing verbs and arguments that don't go together
  - For example
  - VP -> V NP therefore

    Sneezed the book is a VP since "sneeze" is a verb and "the book" is a valid NP

CFG Solution for Agreement

- It works and stays within the power of CFGs
- But its ugly
- And it doesn't scale all that well because of the interaction among the various constraints explodes the number of rules in our grammar.

The Point

- CFGs appear to be just about what we need to account for a lot of basic syntactic structure in English.
- But there are problems
  - That can be dealt with adequately, although not elegantly, by staying within the CFG framework.
- There are simpler, more elegant, solutions that take us out of the CFG framework (beyond its formal power)
  - LFG (lexical function grammar), HPSG (head-driven phrase structure grammar), TAG (Tree-adjoining grammar), CCG (combinatory categorial grammar), etc.

Treebanks

- Treebanks are corpora in which each sentence has been paired with a parse tree (presumably the right one).
- These are generally created
  - By first parsing the collection with an automatic parser
  - And then having human annotators correct each parse as necessary.
- This generally requires detailed annotation guidelines that provide a POS tagset, a grammar and instructions for how to deal with particular grammatical constructions.

Penn Treebank

- Penn TreeBank is a widely used treebank.
  - Other Treebanks are also available

- Most well known is the Wall Street Journal section of the Penn TreeBank.
  - 1 M words from the 1987-1989 Wall Street Journal.

Penn Treebank

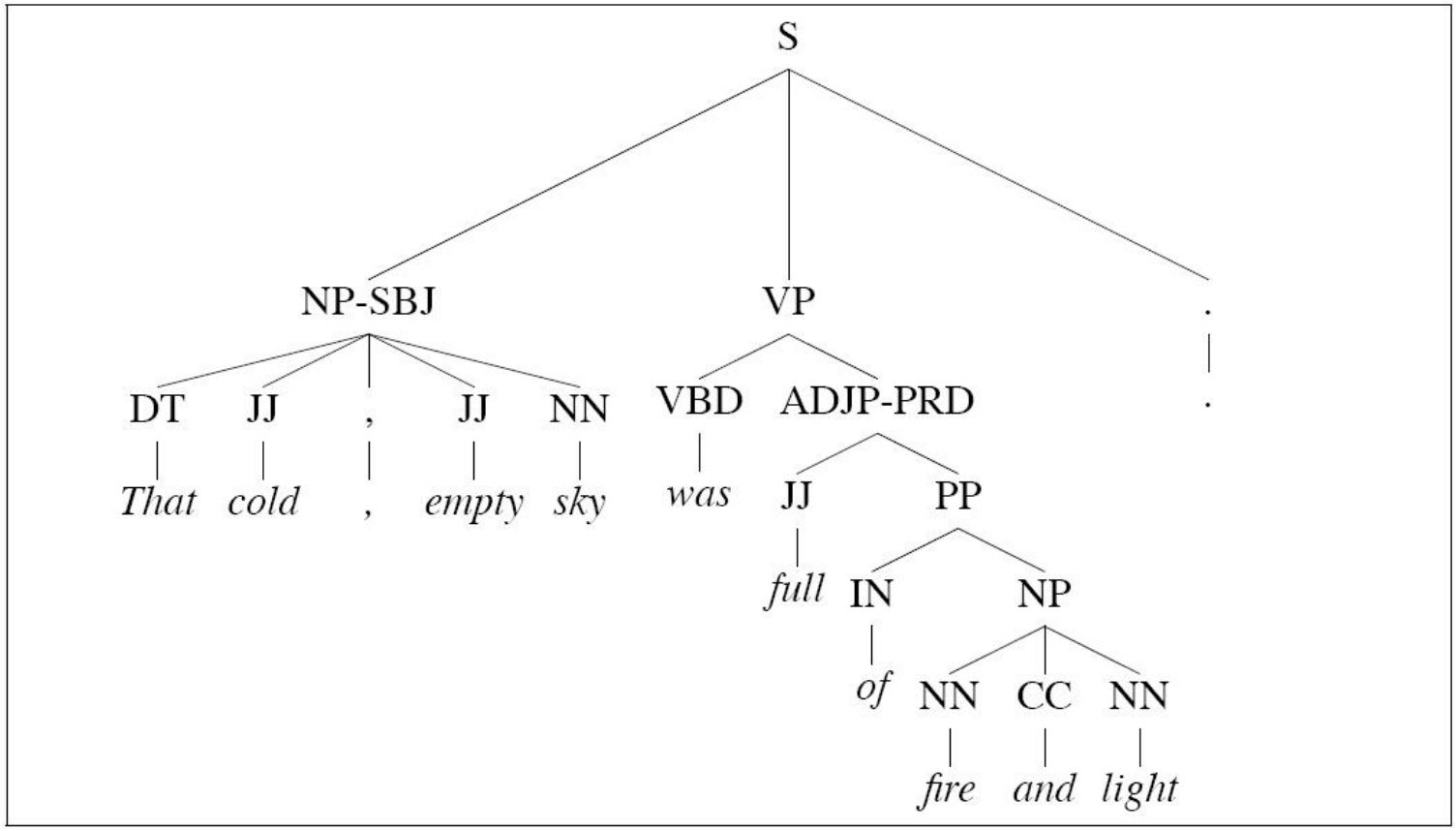
```
((S
   (NP-SBJ (DT That)
     (JJ cold) (, ,)
     (JJ empty) (NN sky) )
   (VP (VBD was)
     (ADJP-PRD (JJ full)
       (PP (IN of)
         (NP (NN fire)
           (CC and)
           (NN light) ))))
   (. .) ))
```

(a)

```
((S
   (NP-SBJ The/DT flight/NN )
   (VP should/MD
     (VP arrive/VB
       (PP-TMP at/IN
         (NP eleven/CD a.m/RB ))
       (NP-TMP tomorrow/NN )))))
```

(b)

Penn Treebank



Penn Treebank

```
( (S (`` ``)
    (S-TPC-2
      (NP-SBJ-1 (PRP We) )
      (VP (MD would)
        (VP (VB have)
          (S
            (NP-SBJ (-NONE- *-1) )
            (VP (TO to)
              (VP (VB wait)
                (SBAR-TMP (IN until)
                  (S
                    (NP-SBJ (PRP we) )
                    (VP (VBP have)
                      (VP (VBN collected)
                        (PP-CLR (IN on)
                          (NP (DT those)(NNS assets)))))))))))))
    (, ,) (`` ``)
    (NP-SBJ (PRP he) )
    (VP (VBD said)
      (S (-NONE- *T*-2) ))
    (. .) ))
```

Treebank Grammars

- Treebanks implicitly define a grammar for the language covered in the treebank.
- Simply take the local rules that make up the sub-trees in all the trees in the collection and you have a grammar.
- Not complete, but if you have decent size corpus, you'll have a grammar with decent coverage.

Treebank Grammars

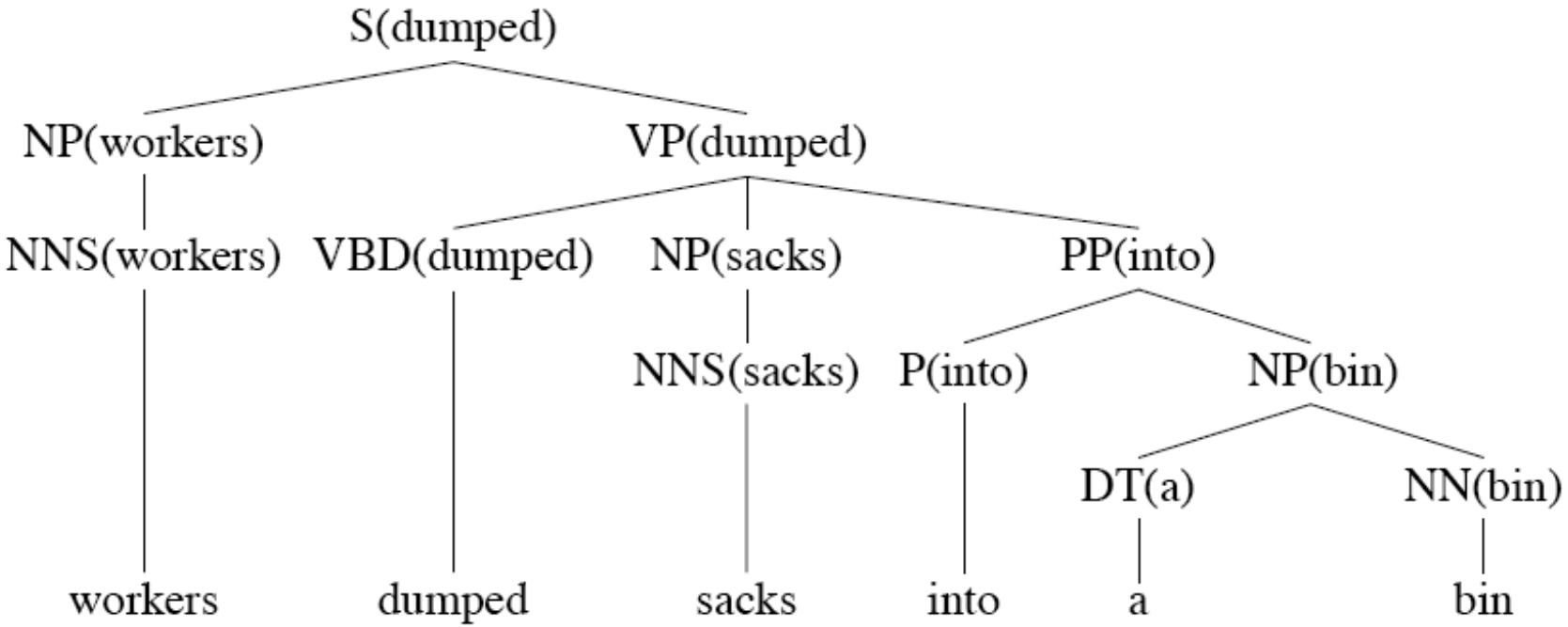
- Such grammars tend to be very flat due to the fact that they tend to avoid recursion.
  - To ease the annotators burden
- For example, the Penn Treebank has 4500 different rules for VPs. Among them...

```
VP → VBD PP
VP → VBD PP PP
VP → VBD PP PP PP
VP → VBD PP PP PP PP
```

Heads in Trees

- Finding heads in treebank trees is a task that arises frequently in many applications.
  - Particularly important in statistical parsing
- We can visualize this task by annotating the nodes of a parse tree with the heads of each corresponding node.

Lexically Decorated Tree



Head Finding

The standard way to do head finding is to use a simple set of tree traversal rules specific to each non-terminal in the grammar.

Parent	Direction	Priority List
ADJP	Left	NNS QP NN $ ADVP JJ VBN VBG ADJP JJR NP JJS DT FW RBR RBS SBAR RB
ADVP	Right	RB RBR RBS FW ADVP TO CD JJR JJ IN NP JJS NN
PRN	Left	
PRT	Right	RP
QP	Left	$ IN NNS NN JJ RB DT CD NCD QP JJR JJS
S	Left	TO IN VP S SBAR ADJP UCP NP
SBAR	Left	WHNP WHPP WHADVP WHADJP IN DT S SQ SINV SBAR FRAG
VP	Left	TO VBD VBN MD VBZ VB VBG VBP VP ADJP NN NNS NP

Treebank Uses

- Treebanks (and headfinding) are particularly critical to the development of statistical parsers
  - Covered later
- Also valuable to *Corpus Linguistics*
  - Investigating the empirical details of various constructions in a given language

Summary

- Context-free grammars can be used to model various facts about the syntax of a language.
- When paired with parsers, such grammars constitute a critical component in many applications.
- Constituency is a key phenomena easily captured with CFG rules.
  - But agreement and subcategorization do pose significant problems
- Treebanks pair sentences in corpus with their corresponding trees.

Parsing with CFG

Syntactic Parsing

- Declarative formalisms like CFGs define the legal strings of a language but don't specify how to recognize or assign structure to them
- Parsing algorithms specify how to recognize the strings of a language and assign each string one or more syntactic structures
- Parse trees useful for grammar checking, semantic analysis, MT, QA, information extraction, speech recognition…and almost every task in NLP

Parsing

- Parsing with CFGs refers to the task of assigning proper trees to input strings
- Proper here means a tree that covers all and only the elements of the input and has an S at the top
- It doesn't actually mean that the system can select the correct tree from among all the possible trees
- As with everything of interest, parsing involves a search which involves the making of choices

Example Grammar

Grammar	Lexicon
$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Pronoun$	$Pronoun \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid NWA$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	$Preposition \rightarrow from \mid to \mid on \mid near \mid through$
$Nominal \rightarrow Nominal\ Noun$	
$Nominal \rightarrow Nominal\ PP$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	
$VP \rightarrow Verb\ NP\ PP$	
$VP \rightarrow Verb\ PP$	
$VP \rightarrow VP\ PP$	
$PP \rightarrow Preposition\ NP$	

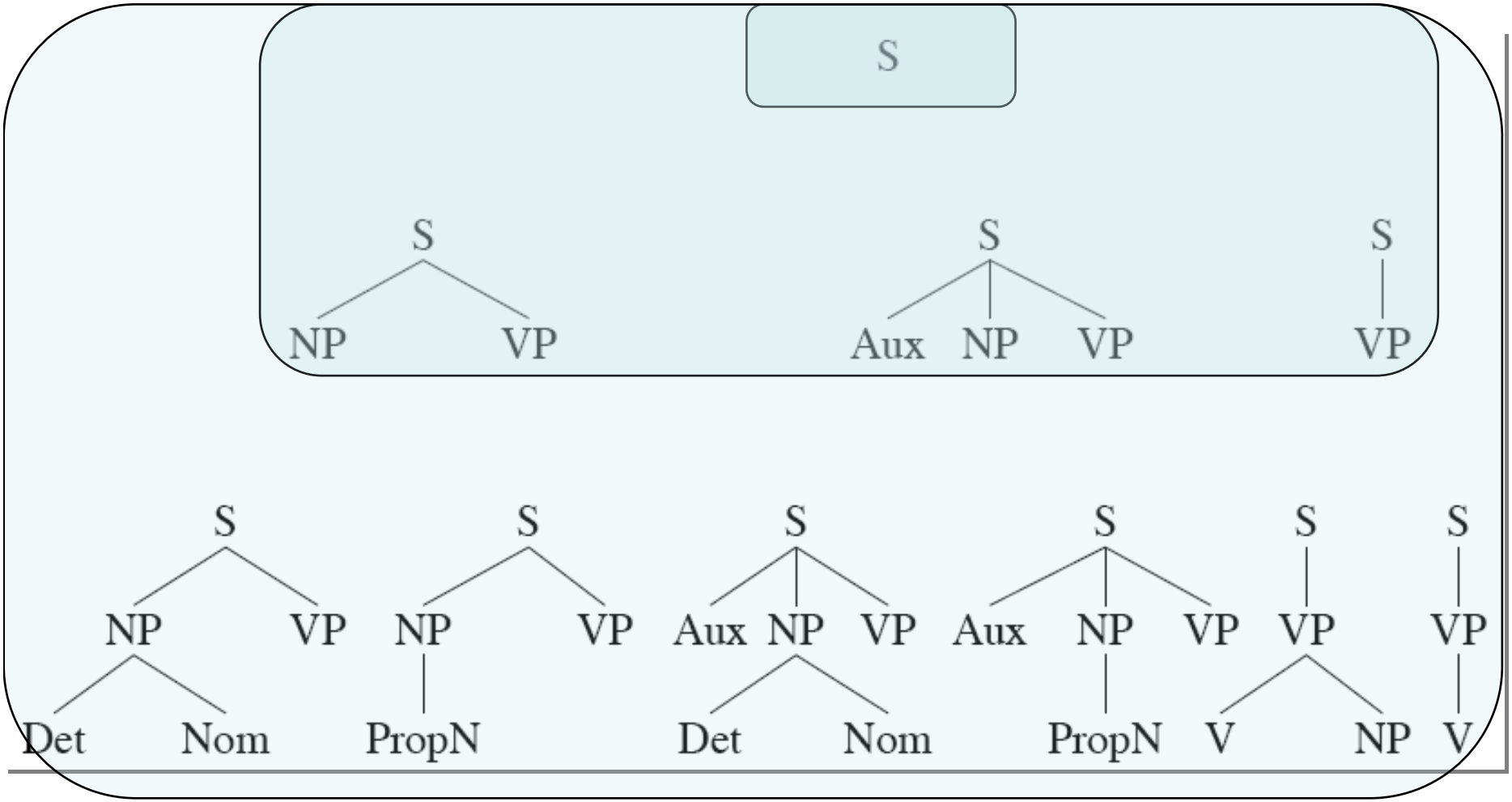
Top-Down Search

- Since we're trying to find trees rooted with an *S* (Sentences), why not start with the rules that give us an *S*.
- Then we can work our way down from there to the words.

Top-Down Parser

- Builds from the root S node to the leaves
- Assuming we build all trees in parallel:
  - Find all trees with root S
  - Next expand all constituents in these trees/rules
  - Continue until leaves are pos
  - Candidate trees failing to match pos of input string are rejected

Top Down Space

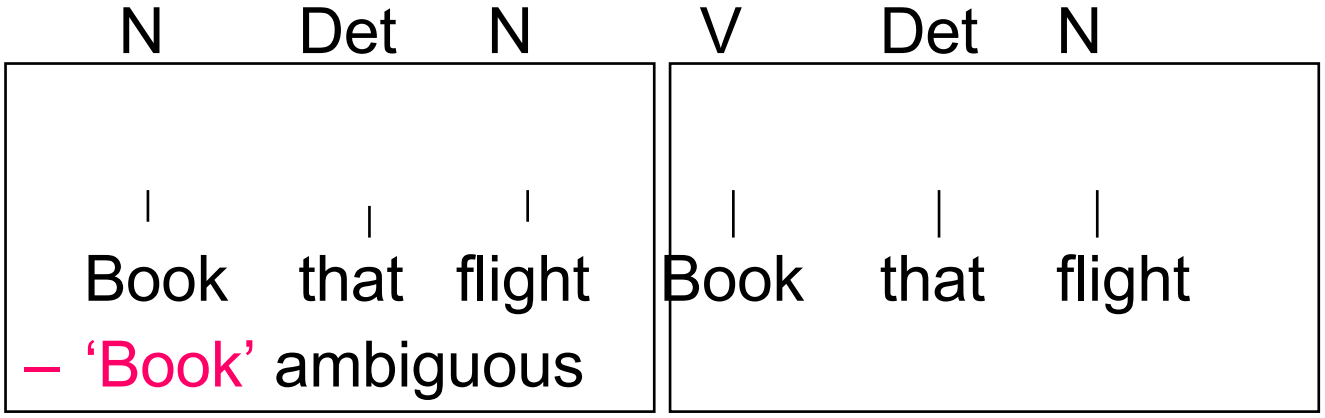


Bottom-Up Parsing

- Of course, we also want trees that cover the input words. So we might also start with trees that link up with the words in the right way.
- Then work your way up from there to larger and larger trees.

Bottom-Up Parsing

- Parser begins with words of input and builds up trees, applying grammar rules w/rhs match
  - Book that flight

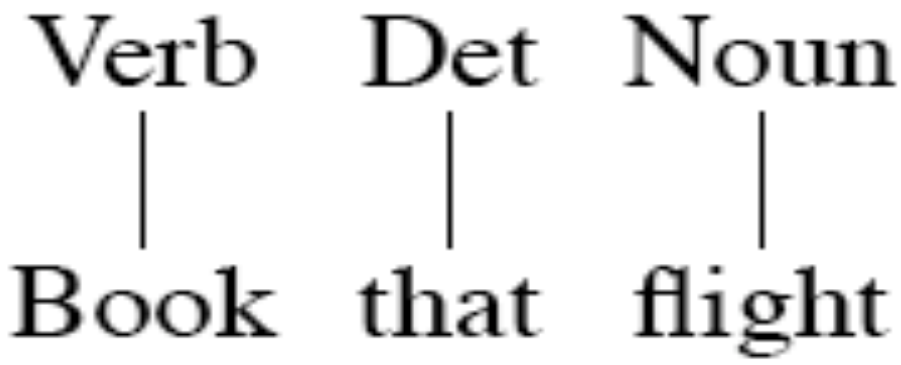


  - ‘Book’ ambiguous
  - Parse continues until an S root node reached or no further node expansion possible

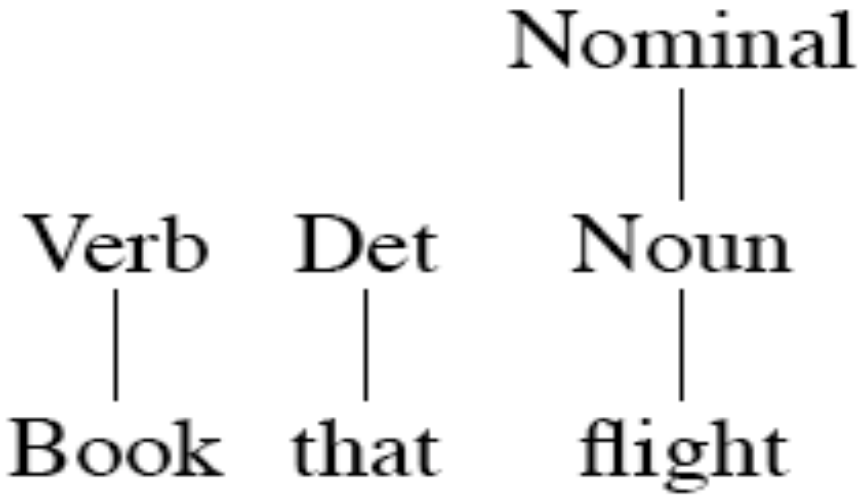
Bottom-Up Search

Book that flight

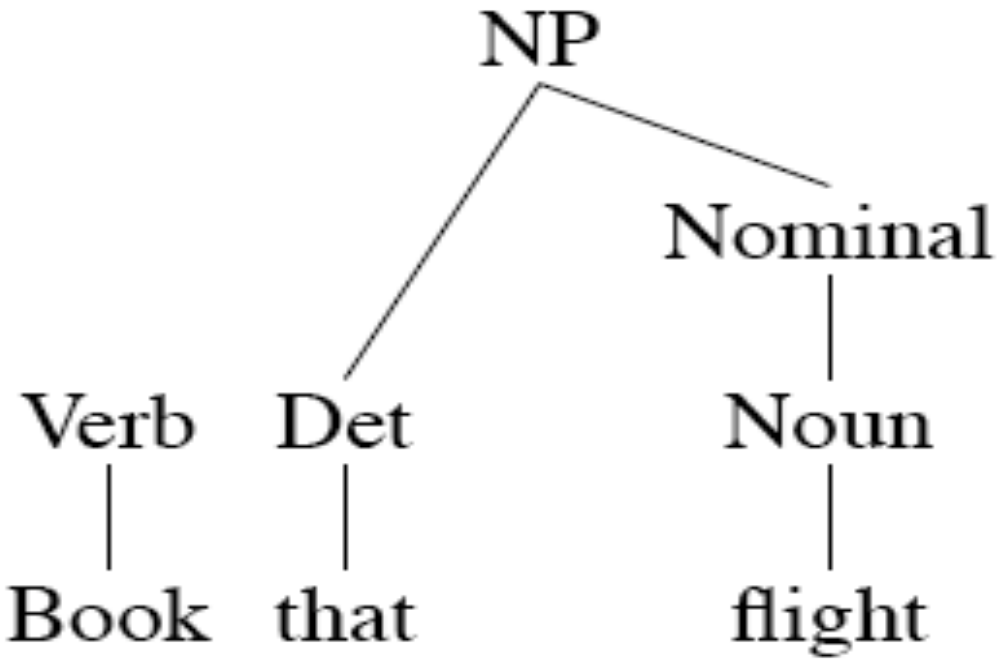
Bottom-Up Search



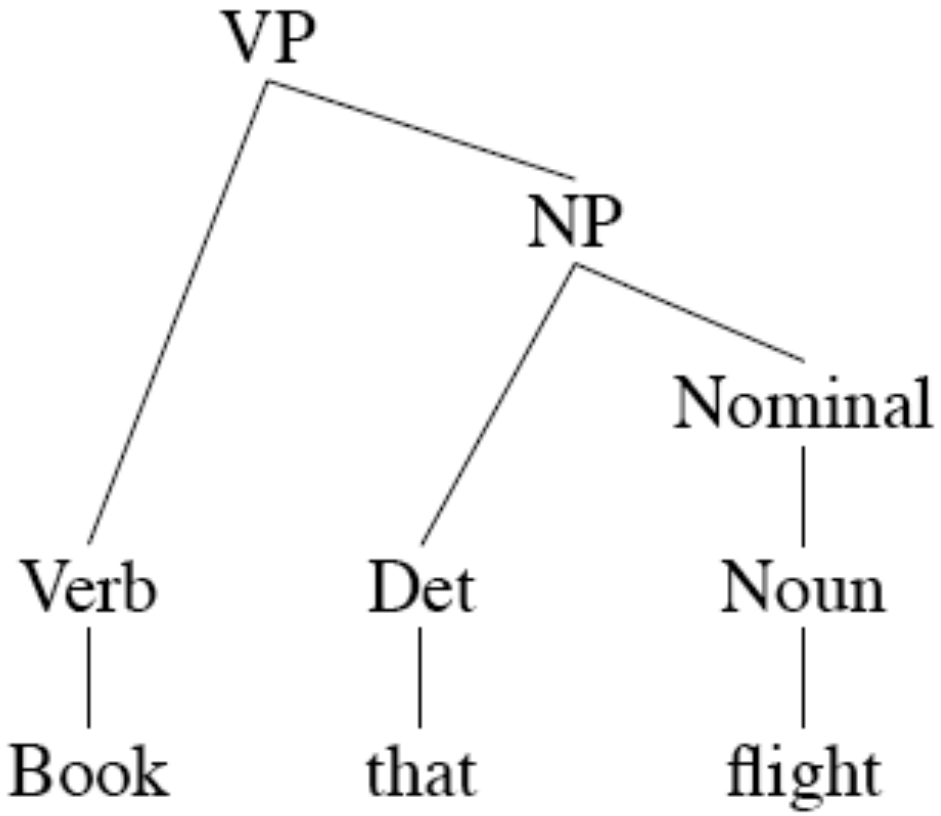
Bottom-Up Search



Bottom-Up Search



Bottom-Up Search



Comparing Top-Down and Bottom-Up

- **Top-Down parsers** never explore illegal parses (e.g. can't form an S) -- but waste time on trees that can never match the input
- **Bottom-Up parsers** never explore trees inconsistent with input -- but waste time exploring illegal parses (no S root)
- In both cases, we assume processing is done in parallel, but it is not practical to do so.
- Search with backtracking

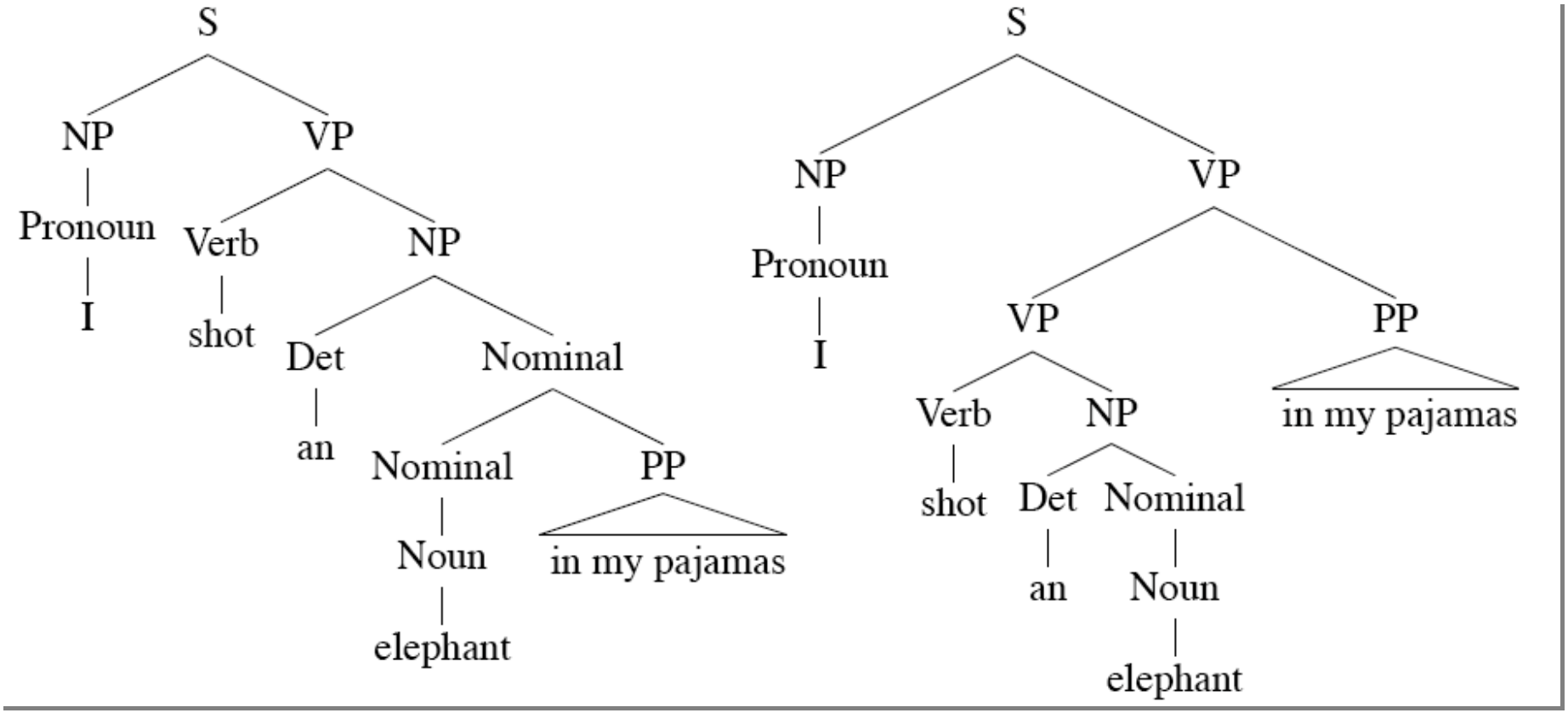
Backtracking

- One approach is called backtracking.
  - Make a choice, if it works out then fine
  - If not then back up and make a different choice
- Backtracking methods are doomed because of two problems
  - Ambiguity
  - Shared subproblems

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

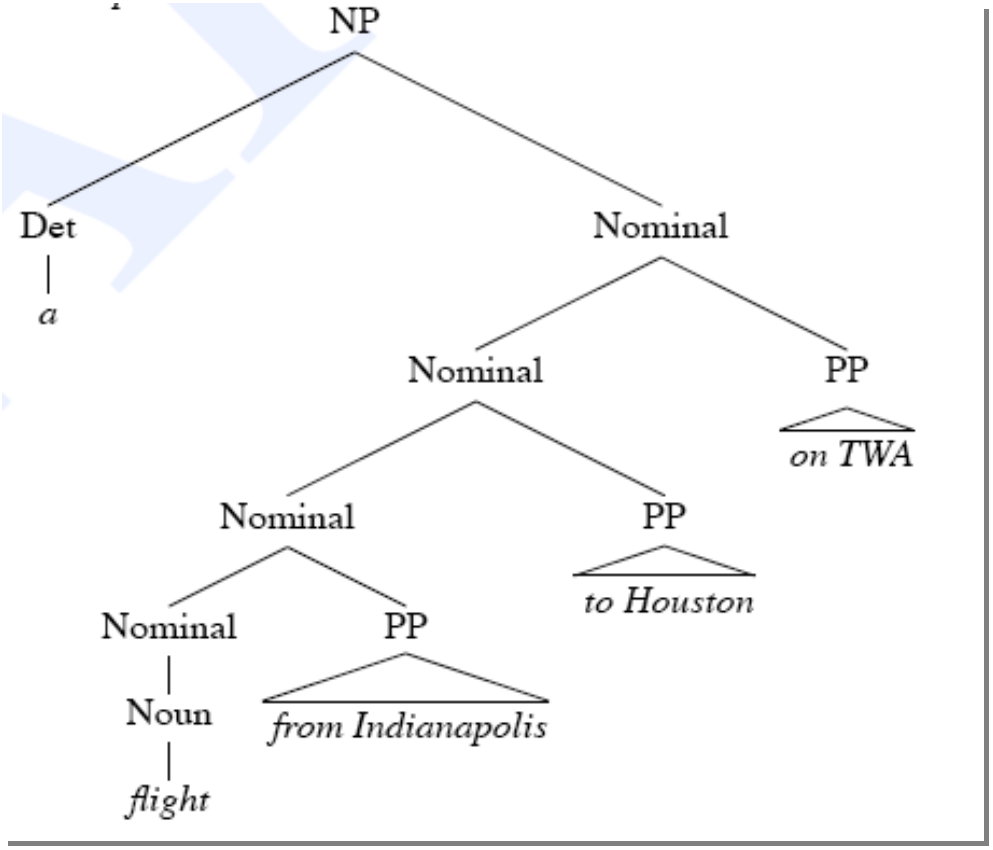

S
NP
VP
Pronoun
Verb
NP
I
shot
Det
Nominal
an
Nominal
PP
Noun
in my pajamas
elephant
S
NP
VP
Pronoun
VP
PP
I
Verb
NP
in my pajamas
shot
Det
Nominal
an
Noun
elephant

Shared Sub-Problems

- No matter what kind of search (top-down or bottom-up or mixed) that we choose.
  - We don't want to redo work we've already done.
  - Unfortunately, naïve backtracking will lead to duplicated work.

Shared Sub-Problems

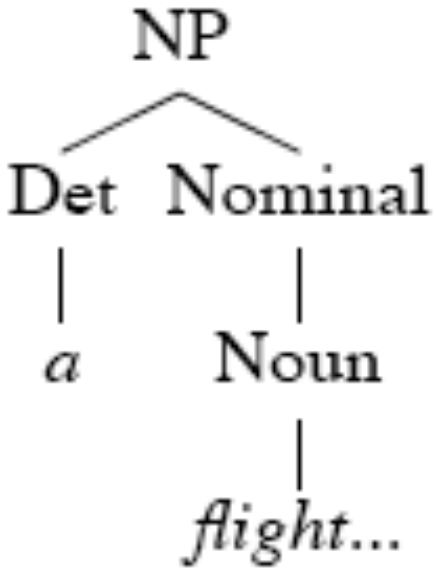
- Consider
  - *A flight from Indianapolis to Houston on TWA*



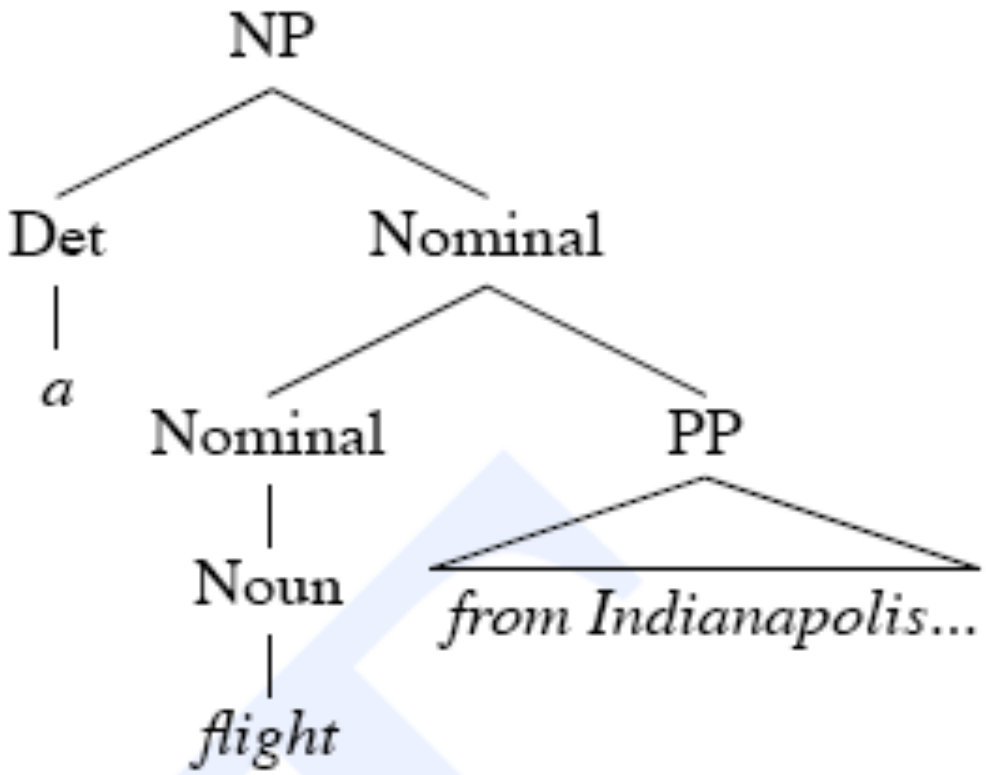
Shared Sub-Problems

- Assume a top-down parse making choices among the various Nominal rules.
- In particular, between these two
  - Nominal -> Noun
  - Nominal -> Nominal PP
- Statically choosing the rules in this order leads to the following bad results...

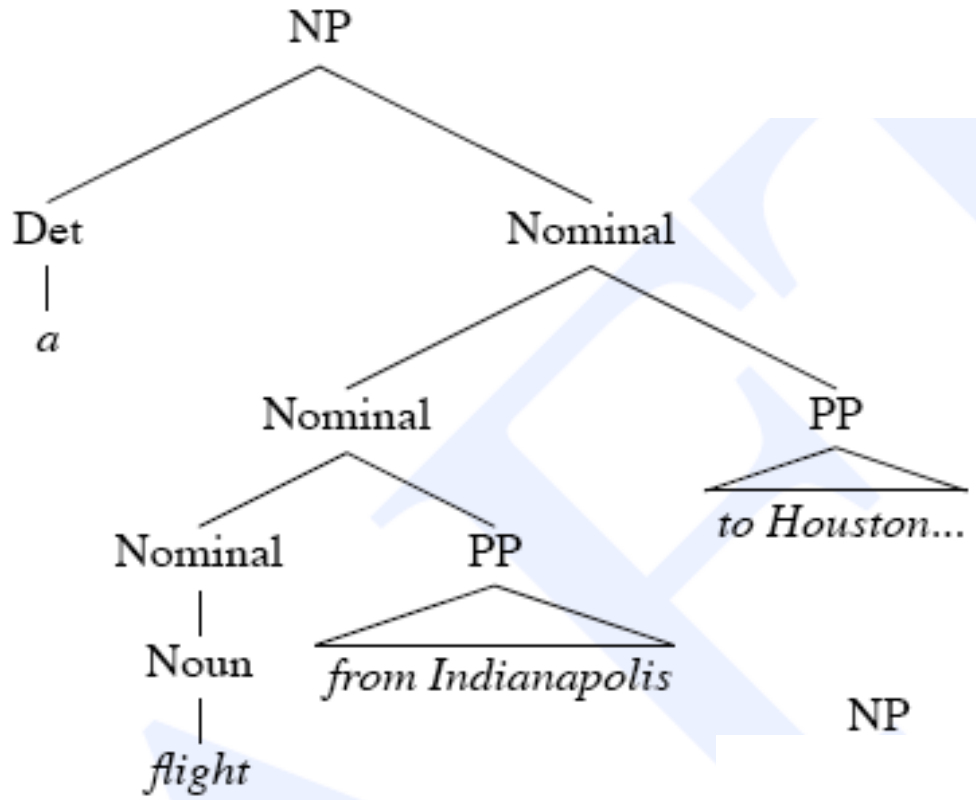
Shared Sub-Problems



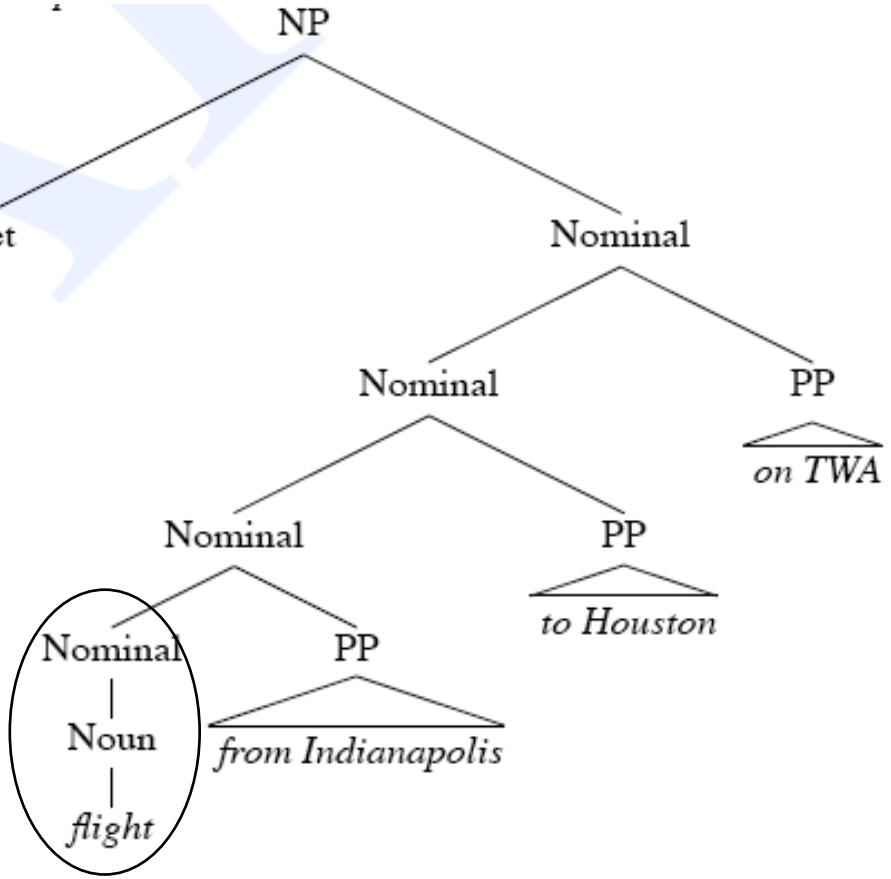
Shared Sub-Problems



Shared Sub-Problems



Shared Sub-Problems



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, avoiding repeated work.
- Since all parses implicitly stored, all available for later disambiguation
- We will look at two approaches corresponding to top-down and bottom-up:
  - CYK: Cocke-Younger-Kasami (CYK) (1960),
  - Earley: Earley (1970)

CKY Parsing

- Limit our grammar to Chomsky Normal Form
  - A $\rightarrow$ BC (two non-terminals)
  - A $\rightarrow$ a (one terminal)
- Consider the rule *A* $\rightarrow$ *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
    - i.e., A is split to B followed by C somewhere.

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.
  - Weak equivalence

Problem

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

  $A \rightarrow B\ C$

  Or

  $A \rightarrow a$

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

Conversion to CNF

- Replace terminals with non-terminals for rules that mix terminals and non-terminals on RHS.
- Eliminate chains of unit productions.
- Introduce new intermediate non-terminals into the grammar that distribute rules with length > 2 over several rules.
  - So… $S \rightarrow A\ B\ C$ *turns into*

  $S \rightarrow X\ C$ *and*

  $X \rightarrow A\ B$

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

Sample Grammar

Grammar	Lexicon
$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Pronoun$	$Pronoun \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid NWA$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	$Preposition \rightarrow from \mid to \mid on \mid near \mid through$
$Nominal \rightarrow Nominal\ Noun$	
$Nominal \rightarrow Nominal\ PP$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	
$VP \rightarrow Verb\ NP\ PP$	
$VP \rightarrow Verb\ PP$	
$VP \rightarrow VP\ PP$	
$PP \rightarrow Preposition\ NP$	

CNF Conversion

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

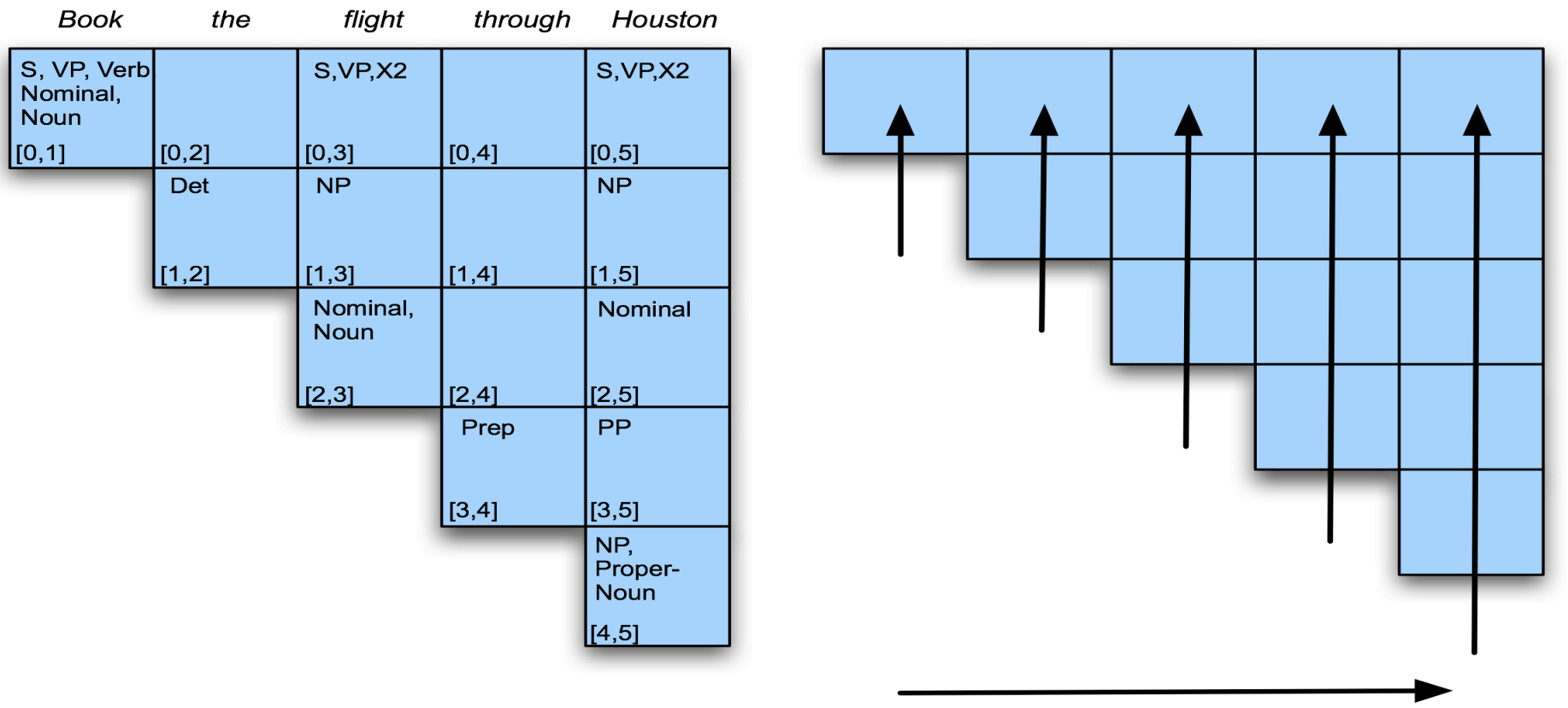
CKY Intuition

- So let's build a table so that each cell [i,j] in the table stores the constituent (e.g., A) spanning from i to j in the input.
- 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.
- In other words, if we think there might be an A spanning i,j in the input then we need to check if there exists a rule $A \rightarrow B\ C$ where B is in [i,k] and C is in [k,j] for some i<k<j

CKY

- 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


Book
the
flight
through
Houston
S, VP, Verb Nominal, Noun
[0,1]
[0,2]
S,VP,X2
[0,3]
[0,4]
S,VP,X2
[0,5]
Det
[1,2]
NP
[1,3]
[1,4]
NP
[1,5]
Nominal, Noun
[2,3]
[2,4]
Nominal
[2,5]
Prep
[3,4]
PP
[3,5]
NP, Proper-Noun
[4,5]

CKY Algorithm

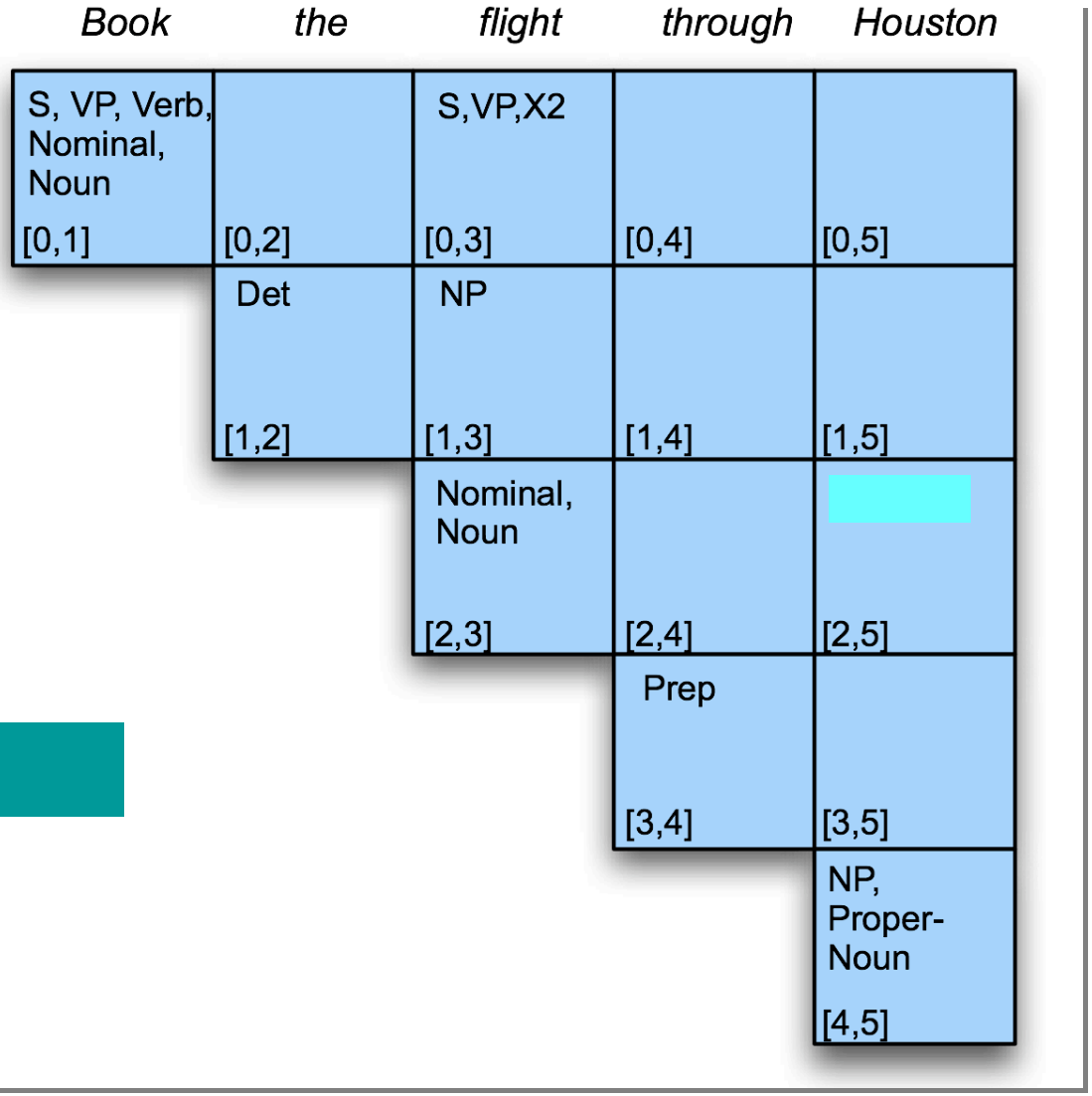
function CKY-PARSE(*words*, *grammar*) **returns** *table*

for $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
  $table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$
  for $i \leftarrow$ **from** $j-2$ **downto** 0 **do**
    for $k \leftarrow i+1$ **to** $j-1$ **do**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k],$
          $C \in table[k,j]\}$

CKY Parsing

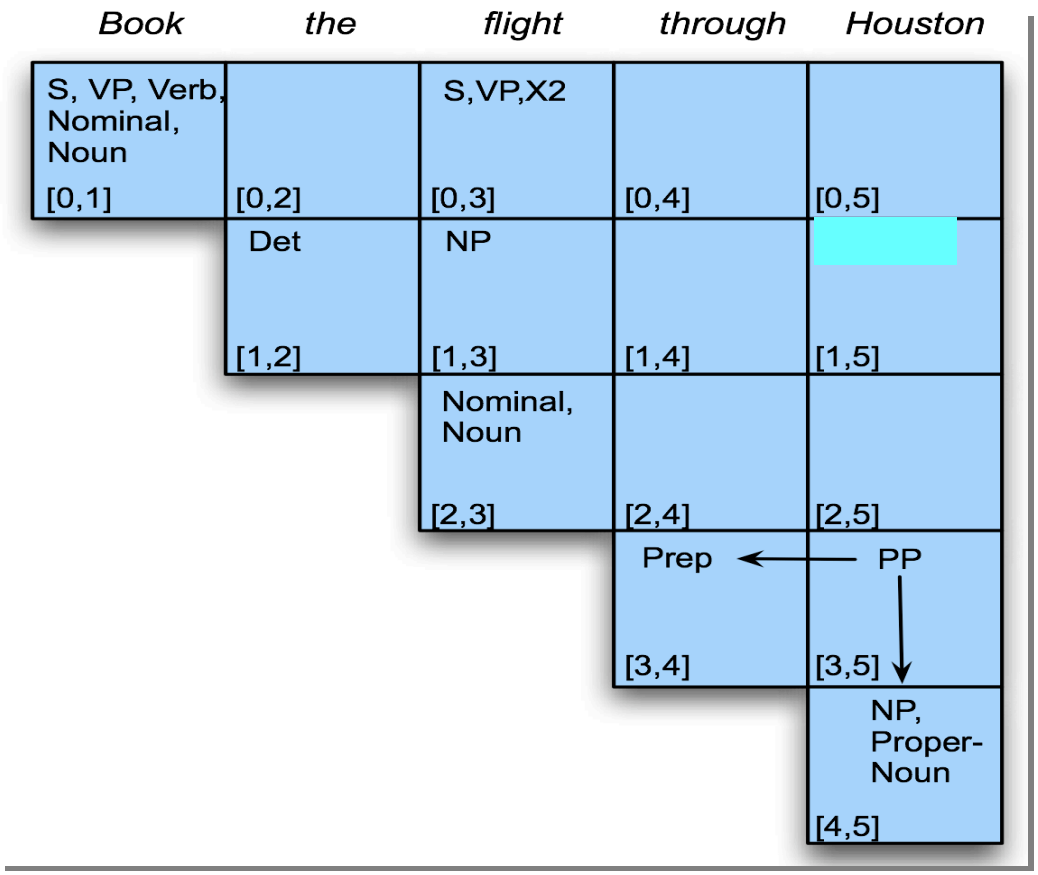
- Is that really a parser?
- What needs to be changed to turn this algorithm into a parser?

Example

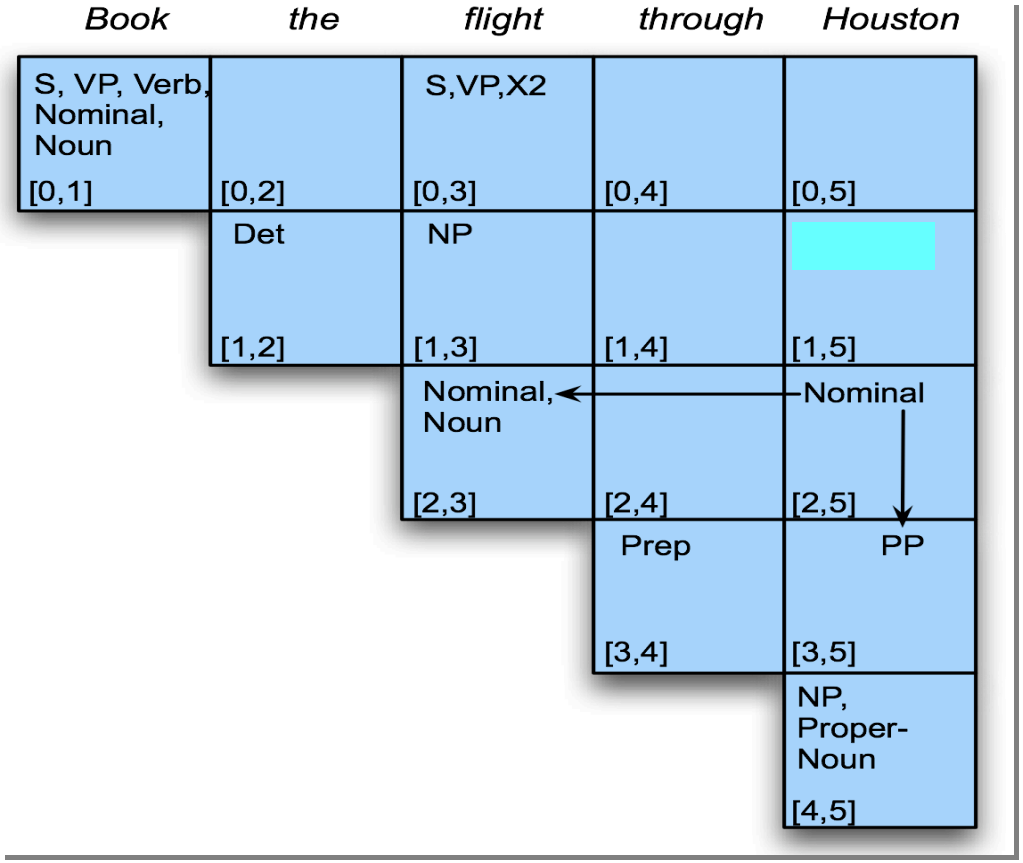


Filling column 5

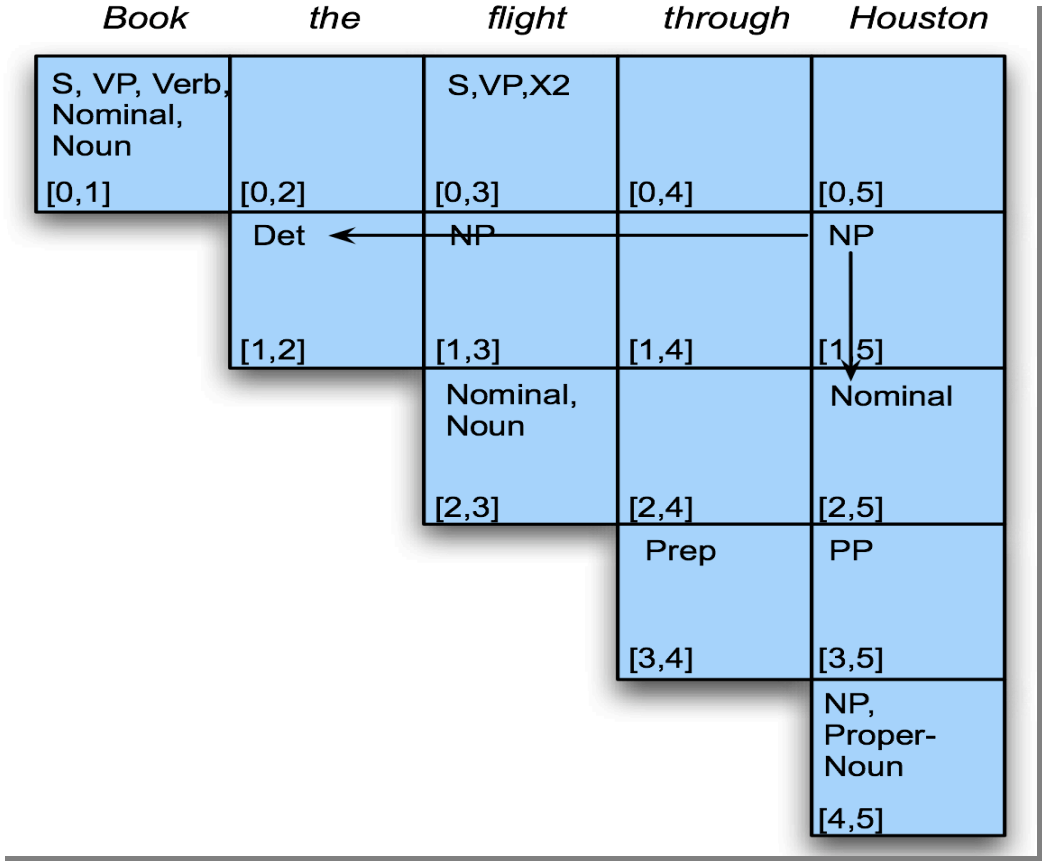
Example


Book
the
flight
through
Houston
S, VP, Verb, Nominal, Noun
[0,1]
[0,2]
S,VP,X2
[0,3]
[0,4]
[0,5]
Det
[1,2]
NP
[1,3]
[1,4]
[1,5]
Nominal, Noun
[2,3]
[2,4]
[2,5]
Prep
PP
[3,4]
[3,5]
NP, Proper-Noun
[4,5]

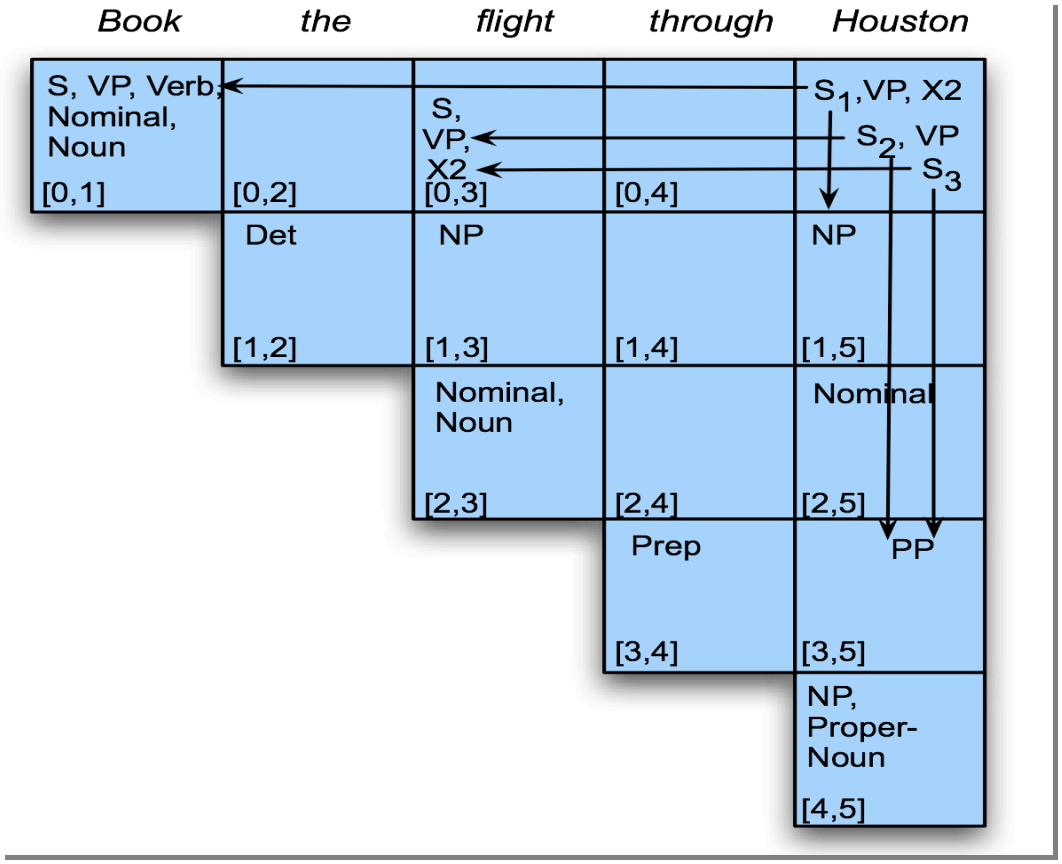
Example


Book
the
flight
through
Houston
S, VP, Verb, Nominal, Noun
[0,1]
[0,2]
S,VP,X2
[0,3]
[0,4]
[0,5]
Det
[1,2]
NP
[1,3]
[1,4]
[1,5]
Nominal, Noun
[2,3]
Nominal
[2,4]
[2,5]
Prep
[3,4]
PP
[3,5]
NP, Proper-Noun
[4,5]

Example


Book
the
flight
through
Houston
S, VP, Verb, Nominal, Noun
[0,1]
[0,2]
S,VP,X2
[0,3]
[0,4]
[0,5]
Det
NP
NP
[1,2]
[1,3]
[1,4]
[1,5]
Nominal, Noun
[2,3]
[2,4]
Nominal
[2,5]
Prep
[3,4]
PP
[3,5]
NP, Proper-Noun
[4,5]

Example


Book
the
flight
through
Houston
S, VP, Verb, Nominal, Noun
[0,1]
[0,2]
S, VP, X2
[0,3]
[0,4]
S1, VP, X2
S2, VP
S3
Det
[1,2]
NP
[1,3]
[1,4]
NP
[1,5]
Nominal, Noun
[2,3]
[2,4]
Nominal
[2,5]
Prep
[3,4]
PP
[3,5]
NP, Proper-Noun
[4,5]

CKY Summary

- The problems of constructing parsing trees using CKY in practice.
  - Post-process resulting trees for restoration.
- 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 of a given sentence
  - To avoid this we can switch to a top-down control strategy

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
  - 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.
    - Completed constituents and their locations
    - In-progress constituents
    - Predicted constituents

States

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

$S \rightarrow \cdot\ VP$	A VP is predicted
$NP \rightarrow Det \cdot\ Nominal$	An NP is in progress
$VP \rightarrow V\ NP \cdot$	A VP has been found

States/Locations

–[x,y] tells us where the state begins (x) and where the dot lies (y) with respect to the input

- S → •VP [0,0]
- NP → Det• Nominal [1,2]
- VP → V NP • [0,3]

- A VP is predicted at the start of the sentence
- An NP is in progress; the Det goes from 1 to 2
- 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 here too
- So, this is a top-down prediction

NP --> Det • Nom, [1,2]

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

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)

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)
```

Predictor

- Intuition: create new states representing 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

  V --> book • [0,1]

Completer

- Intuition: parser has discovered a constituent, so must find and advance all states that are 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]

Earley Code

procedure PREDICTOR($(A \rightarrow \alpha \bullet B\ \beta,\ [i,j])$)
  for each $(B \rightarrow \gamma)$ **in** GRAMMAR-RULES-FOR(B, *grammar*) **do**
    ENQUEUE($(B \rightarrow \bullet\ \gamma,\ [j,j])$, *chart*[*j*])
  end

procedure SCANNER($(A \rightarrow \alpha \bullet B\ \beta,\ [i,j])$)
  if B $\subset$ PARTS-OF-SPEECH(*word[j]*) **then**
    ENQUEUE($(B \rightarrow word[j],\ [j, j+1])$, *chart[j+1]*)

procedure COMPLETER($(B \rightarrow \gamma \bullet,\ [j,k])$)
  for each $(A \rightarrow \alpha \bullet B\ \beta,\ [i,j])$ **in** *chart*[*j*] **do**
    ENQUEUE($(A \rightarrow \alpha\ B \bullet \beta,\ [i,k])$, *chart[k]*)
  end

CFG for Fragment of English

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

Note: the example here uses less rules than the example in the book.

Book that flight (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

- 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 does not need to add V --> Book •, [0,1] again to Chart[1]

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] whose left corner is V and adds them to Chart[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….

Chart[2]

Det →that •	[1,2]	Scanner
NP → Det • Nom	[1,2]	Completer
Nom → • N	[2,2]	Predictor
Nom→ • Nom N	[2,2]	Predictor

Chart[3]

N → flight •	[2,3]	Scanner
Nom → N •	[2,3]	Completer
NP → Det Nom •	[1,3]	Completer
Nom → Nom • N	[2,3]	Completer
VP → V NP •	[0,3]	Completer
S → VP •	[0,3]	Completer

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, avoiding repeated work.
- Since all parses implicitly stored, all available for later disambiguation
- We have looked at two approaches corresponding to top-down and bottom-up:
  - CYK: Cocke-Younger-Kasami (CYK) (1960),
  - Earley: Earley (1970)

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($(A \rightarrow \alpha \bullet B\ \beta,\ [i,j])$)
  for each $(B \rightarrow \gamma)$ **in** GRAMMAR-RULES-FOR($B, grammar$) **do**
    ENQUEUE($(B \rightarrow \bullet\ \gamma,\ [j,j]), chart[j]$)
  end

procedure SCANNER($(A \rightarrow \alpha \bullet B\ \beta,\ [i,j])$)
  if B $\subset$ PARTS-OF-SPEECH($word[j]$) **then**
    ENQUEUE($(B \rightarrow word[j],\ [j, j+1]), chart[j+1]$)

procedure COMPLETER($(B \rightarrow \gamma \bullet,\ [j,k])$)
  for each $(A \rightarrow \alpha \bullet B\ \beta,\ [i,j])$ **in** $chart[j]$ **do**
    ENQUEUE($(A \rightarrow \alpha\ B \bullet \beta,\ [i,k]), chart[k]$)
  end

How do we retrieve the parses at the end?

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

Efficiency

- For such a simple example, there seems to be a lot of useless stuff in there.
- Why?

• It's predicting things that aren't consistent with the input

•That's the flipside to the CKY problem.

Ambiguity

- Did we solve the ambiguity problem?
- No…
  - Both CKY and Earley will likely 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.

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?

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

Partial Parsing

- Many applications (e.g., IE, IR) only requires partial parse or shallow parse
- Chunking:
  - Flat, non-overlapping segments, non-recursive phrases
  - NP, VP, PP, etc.
  - In most approaches, base phrases include the headword of the phrase, prehead material, while excluding any post-head material.

[**NP** The morning flight] [**PP** from] [**NP** Denver] [**VP** has arrived.]

[**NP** The morning flight] from [**NP** Denver] has arrived.

Finite-State Rule-based Chunking

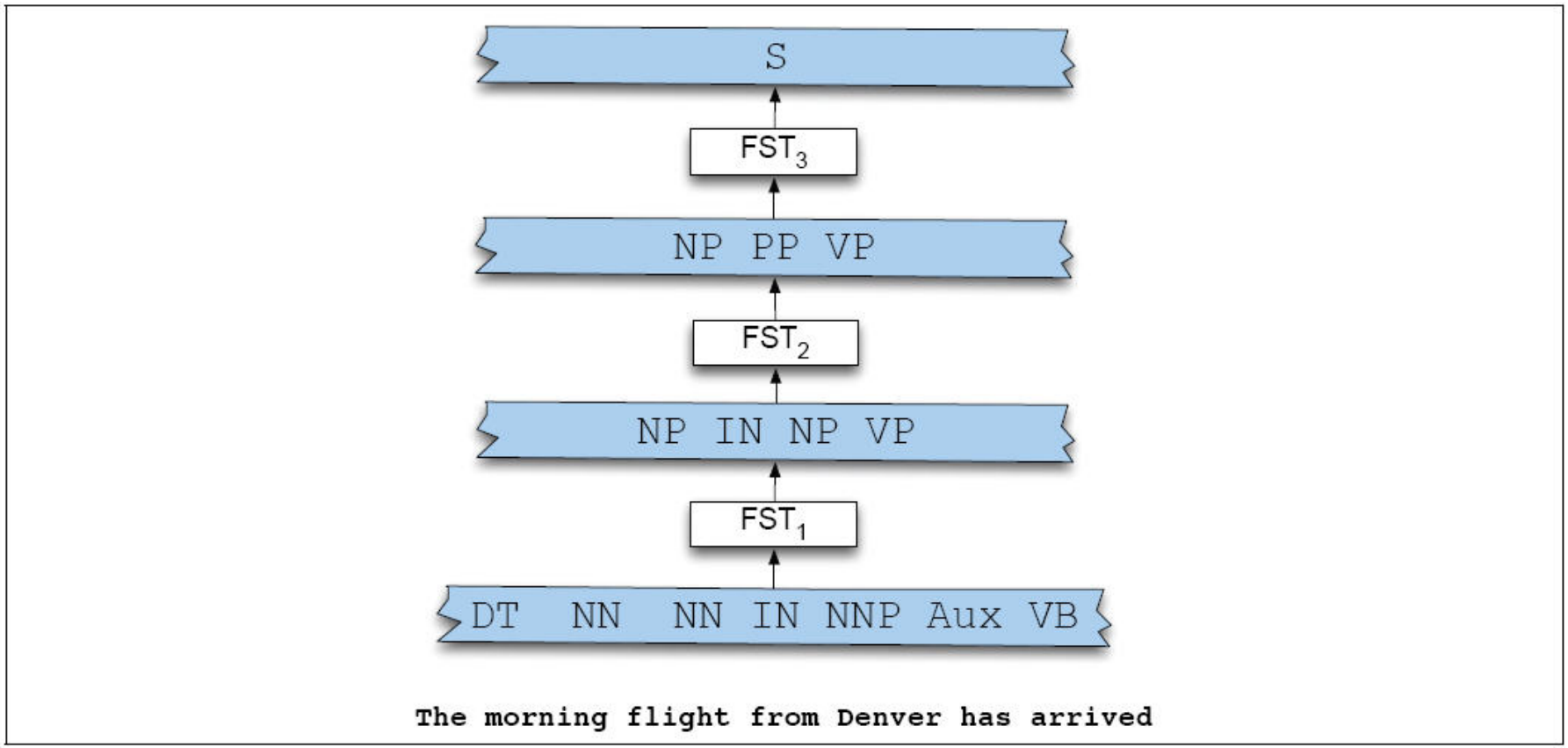
- Finite-state-rules (or regular expressions) to capture constituents.

  NP -> (DT) NN* NN
- Finite-state cascades:

  FST2: PP -> IN NP

  FST3: S->PP*NP PP*VP PP*

Finite-State Rule-based Chunking



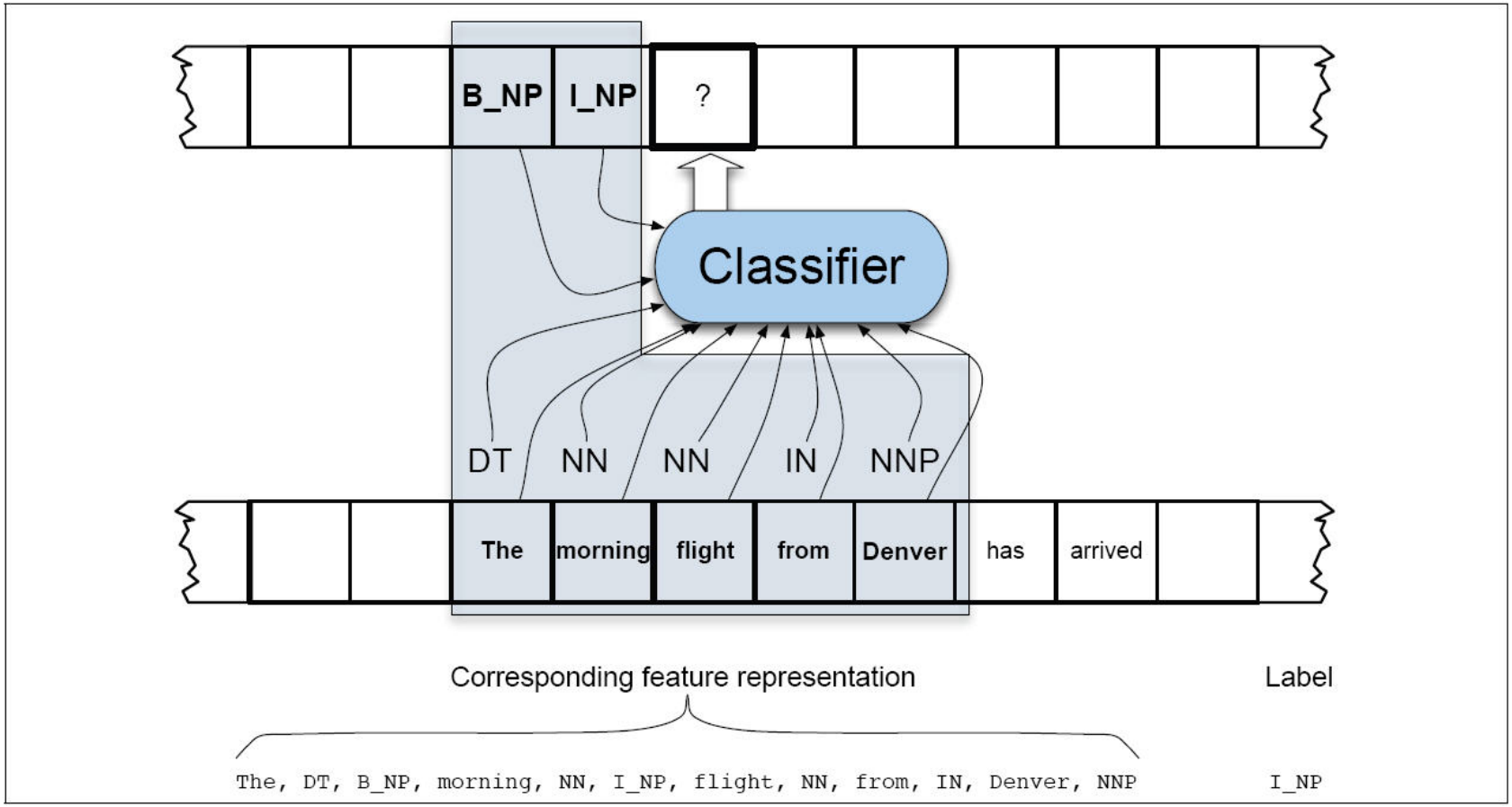
Machine Learning Approaches to Chunking

- Treat chunking as sequential classification problem -> tagging task (similar to POS tagging)
- IOB tagging
  - B: beginning of a segment
  - I: internal of a segment
  - O: outside any chunk

The	morning	flight	from	Denver	has	arrived.
B_NP	I_NP	I_NP	B_PP	B_NP	B_VP	I_VP

The	morning	flight	from	Denver	has	arrived.
B_NP	I_NP	I_NP	O	B_NP	O	O

Machine Learning Approaches to Chunking



Probabilistic Context Free Grammars

- The simplest augmentation of the Context Free Grammar (CFG) is the Probabilistic Context Free Grammar (PCFG)
- CFG is defined by (N, Σ, P, S)

N a set of **non-terminal symbols** (or **variables**)

Σ a set of **terminal symbols** (disjoint from N)

R a set of **rules** or productions, each of the form $A \rightarrow \beta \ [p]$,
where A is a non-terminal,
β is a string of symbols from the infinite set of strings $(\Sigma \cup N)*$,
and p is a number between 0 and 1 expressing $P(\beta|A)$

S a designated **start symbol**

PCFG Augmentation

- Each rule in P is assigned a conditional probability

  $A \rightarrow \beta$ [p]

- A PCFG is a 5-tuple G=(N, Σ, P, S, D) where D is a function assigning probabilities to each rule in P

  P($A \rightarrow \beta$) or P($A \rightarrow \beta$ | A)

- Given a non-terminal A, and r_1, r_2, …, r_i, all expanding A $\sum_{j=1}^{i} P(r_j) = 1$

Example

Grammar		Lexicon
$S \rightarrow NP\ VP$	[.80]	$Det \rightarrow that\ [.10] \mid a\ [.30] \mid the\ [.60]$
$S \rightarrow Aux\ NP\ VP$	[.15]	$Noun \rightarrow book\ [.10] \mid flight\ [.30]$
$S \rightarrow VP$	[.05]	$\mid meal\ [.15] \mid money\ [.05]$
$NP \rightarrow Pronoun$	[.35]	$\mid flights\ [.40] \mid dinner\ [.10]$
$NP \rightarrow Proper\text{-}Noun$	[.30]	$Verb \rightarrow book\ [.30] \mid include\ [.30]$
$NP \rightarrow Det\ Nominal$	[.20]	$\mid prefer;[.40]$
$NP \rightarrow Nominal$	[.15]	$Pronoun \rightarrow I\ [.40] \mid she\ [.05]$
$Nominal \rightarrow Noun$	[.75]	$\mid me\ [.15] \mid you\ [.40]$
$Nominal \rightarrow Nominal\ Noun$	[.20]	$Proper\text{-}Noun \rightarrow Houston\ [.60]$
$Nominal \rightarrow Nominal\ PP$	[.05]	$\mid NWA\ [.40]$
$VP \rightarrow Verb$	[.35]	$Aux \rightarrow does\ [.60] \mid can\ [40]$
$VP \rightarrow Verb\ NP$	[.20]	$Preposition \rightarrow from\ [.30] \mid to\ [.30]$
$VP \rightarrow Verb\ NP\ PP$	[.10]	$\mid on\ [.20] \mid near\ [.15]$
$VP \rightarrow Verb\ PP$	[.15]	$\mid through\ [.05]$
$VP \rightarrow Verb\ NP\ NP$	[.05]	
$VP \rightarrow VP\ PP$	[.15]	
$PP \rightarrow Preposition\ NP$	[1.0]	

Algorithms for PCFG

Given a PCFG and a sentence S, define τ(S) to be the set of trees with S as the yield.

- How do we find the best parse tree for the sentence S?

$$\arg\max_{T \in \tau(S)} P(T, S)$$

- How do we find the probability of S?

$$P(S) = \sum_{T \in \tau(S)} P(T, S)$$

Using Probabilities

- To estimate probabilities concerning a sentence and its parse trees
- Useful in disambiguation
- How can we define the probability of a tree?

$$P(T,S) = \prod_{u \in T} P(r(u))$$

P(T, S) = P(T) P(S | T) = P(T)

Use in Disambiguation

- The most likely tree for a sentence S

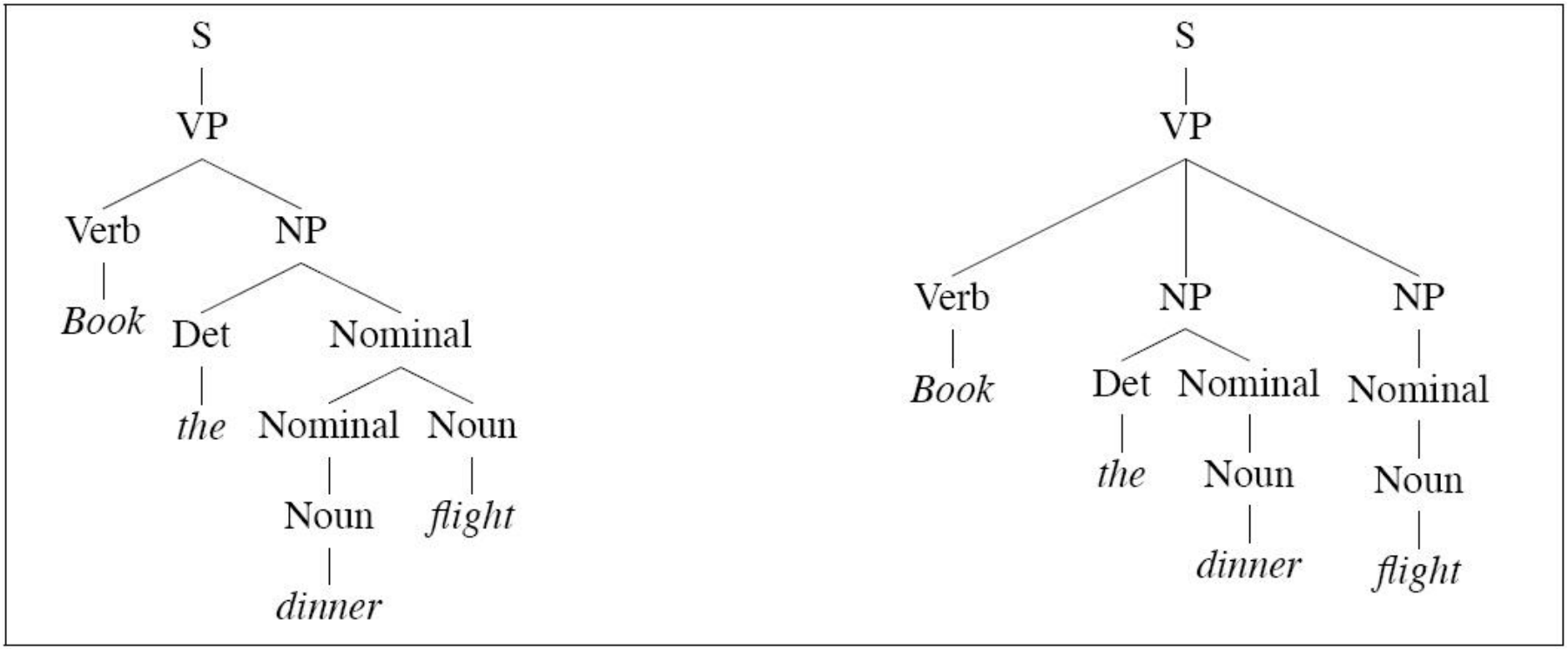
$$\hat{T}(S) = \arg\max_{T \in \tau(S)} P(T \mid S)$$

$$\hat{T}(S) = \arg\max_{T \in \tau(S)} \frac{P(T,S)}{P(S)}$$

$$\hat{T}(S) = \arg\max_{T \in \tau(S)} P(T,S)$$

But P(T, S) = P(T) ⟹ $\hat{T}(S) = \arg\max_{T \in \tau(S)} P(T)$

Disambiguation Example



Disambiguation Example

Rules			P	Rules			P
S	→	VP	.05	S	→	VP	.05
VP	→	Verb NP	.20	VP	→	Verb NP NP	.10
NP	→	Det Nominal	.20	NP	→	Det Nominal	.20
Nominal	→	Nominal Noun	.20	NP	→	Nominal	.15
Nominal	→	Noun	.75	Nominal	→	Noun	.75
				Nominal	→	Noun	.75
Verb	→	book	.30	Verb	→	book	.30
Det	→	the	.60	Det	→	the	.60
Noun	→	dinner	.10	Noun	→	dinner	.10
Noun	→	flights	.40	Noun	→	flights	.40

$P(T_{left}) = 0.05*0.2*0.2*0.2*0.75*0.3*0.6*0.1*0.4 = 2.2 \times 10^{-6}$

$P(T_{right}) = 0.05*0.1*0.2*0.15*0.75*0.75*0.3*0.6*0.1*0.4 = 6.1 \times 10^{-7}$

Use in Language Modeling

- The probability of a sentence S:

$$P(S) = \sum_{T \in \tau(S)} P(T, S) = \sum_{T \in \tau(S)} P(T)$$

- N-gram models can potentially lose important cues from the context:
  - The contract *ended* with a loss of 7 cents *after* trading as low as 9 cents
- Statistical parsers can take advantage of longer-distance information than N-grams.

Learning PCFG Probabilities

Given a set of examples trees, use MLE

$$P(\alpha \to \beta \mid \alpha) = \frac{Count(\alpha \to \beta)}{\sum_{\gamma} Count(\alpha \to \gamma)} = \frac{Count(\alpha \to \beta)}{Count(\alpha)}$$

What if there is no example trees?

- Use inside-outside algorithm (a special case of EM algorithm)

Probabilistic Parsing

- Probabilistic CKY Algorithm: bottom-up parser
- Input:
  - A Chomsky normal form PCFG, G= (N, Σ, P, S, D) Assume that the K non-terminals have indices *1, 2, .., K,* and the start symbol S has index 1
  - *n* words $w_1,\ldots, w_n$
- Data Structure:
  - A dynamic programming array $\pi[i,j,k]$ holds the ***maximum probability*** for a constituent with non-terminal index N_k spanning words *i..j*.
- Output: The maximum probability parse $\pi[0,n,1]$

Review: Chomsky Normal Form

A context free grammar G = (N, Σ, P, S) in Chomsky Normal Form is as follows

- N is a set of non-terminal symbols
- Σ is a set of terminal symbols
- P is a set of rules which take one of two forms
  - $X \rightarrow Y_1 Y_2$ for $X \in N$, and $Y_1, Y_2 \in N$
  - $X \rightarrow Y$ for $X \in N$, and $Y \in \Sigma$
- $S \in N$ is a distinguished start symbol

Base Case

- CKY fills out $\pi[i,j,k]$ by induction
- Base case
  - Input strings with length = 1 (individual words w_i)
  - In CNF, the probability of a given non-terminal N_k expanding to a single word w_i must come only from the rule N_k ->w_j

For all j = 1.. n, for k = 1..K

$$\pi[j-1, j, k] = P(N_k \rightarrow w_j \mid N_k)$$

Recursive Case

- For strings of words > 1, $N_k \Rightarrow^* w_{ij}$ if and only if there is one rule $N_k \rightarrow N_l N_m$ and s, $i \leq s \leq j$ such that N_l derives the first s-i symbols and $N_m$derives the last j-s symbols.
- Since w_{is} and w_{sj} are shorter than w_{ij}, $\pi[i,s,l]$ and $\pi[s+1,j,m]$ are already stored.

$$\pi[i,j,k] = \pi[i,s,l] * \pi[s+1,j,m]$$

$$for\ all\ \ j = 1..n,\ \ i = (j-2)\ downto\ \ 0, k = 1..K,$$

$$\pi[i,j,k] = \max_{\substack{i \leq s < j \\ 1 \leq l \leq K \\ 1 \leq m \leq K}} \{P(N_k \rightarrow N_l N_m \mid N_k) \times \pi[i,s,l] \times \pi[s+1,j,m]\}$$

CKY Algorithm for Probabilistic Parsing

Initialization

for *j* = 1..n, k = 1..K

$$\pi[j-1, j, k] = P(N_k \rightarrow w_j \mid N_k)$$

Main Loop:

for *j=1 .. length*

  for *i = j-2 downto 0*

    for *k = 1..K*

    max = 0;

    for *s = (i+1)...(j-1), l = 1..K, m= 1..K,*

$$prob = P(N_k \rightarrow N_l N_m \mid N_k) \times \pi[i, s, l] \times \pi[s+1, j, m]$$

      if *prob > max*

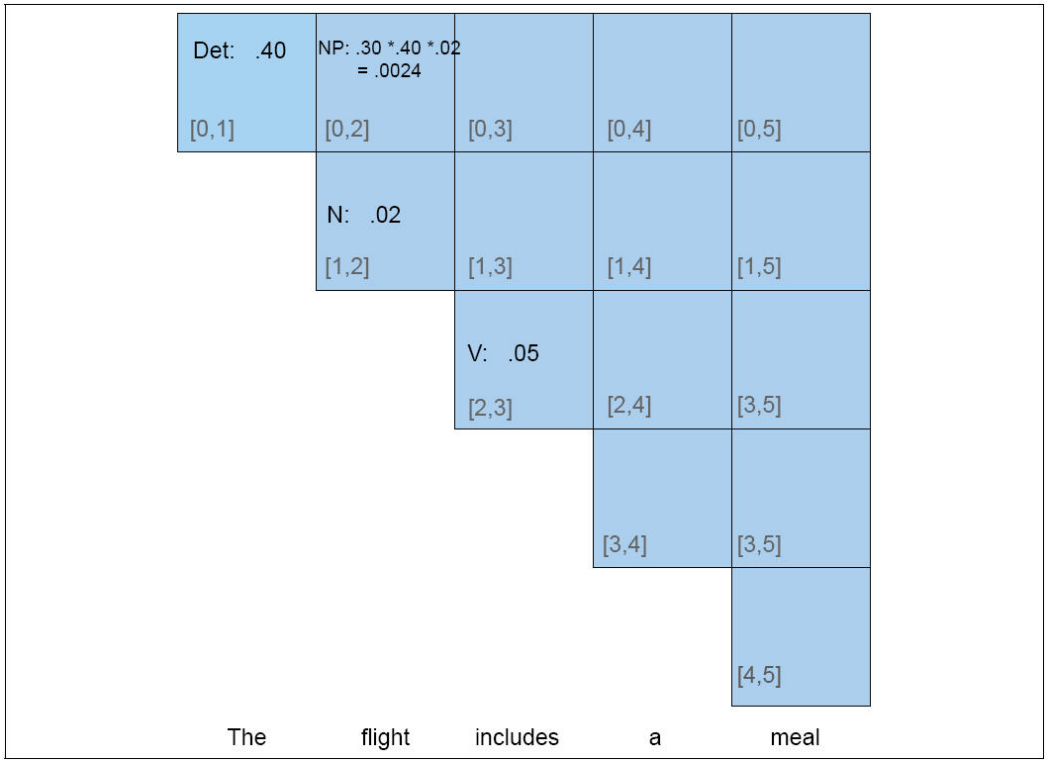
        max = prob;

        split (i, j, k) = {s, l,,m}; //store backpointers to indicate the best split

    π[i,j,k] = max

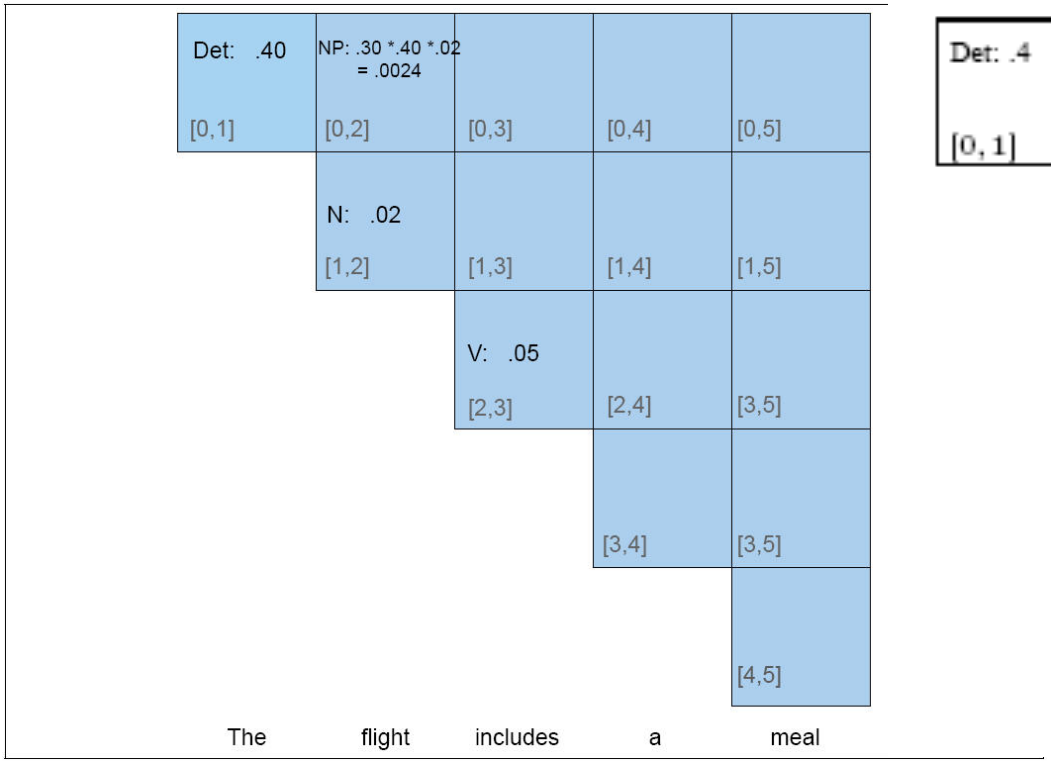
An Example

S	$\rightarrow$	$NP\ VP$	.80	Det	$\rightarrow$	the	.40
NP	$\rightarrow$	$Det\ N$	.30	Det	$\rightarrow$	a	.40
VP	$\rightarrow$	$V\ NP$	.20	N	$\rightarrow$	$meal$	.01
V	$\rightarrow$	$includes$	.05	N	$\rightarrow$	$flight$	.02



An Example

S	$\rightarrow$	$NP\ VP$	.80	Det	$\rightarrow$	the	.40
NP	$\rightarrow$	$Det\ N$	.30	Det	$\rightarrow$	a	.40
VP	$\rightarrow$	$V\ NP$	.20	N	$\rightarrow$	$meal$	.01
V	$\rightarrow$	$includes$	.05	N	$\rightarrow$	$flight$	.02



Det: .4 [0, 1]	NP: .0024 [0, 2]	[0, 3]	[0, 4]	S: 2.304e-8 [0, 5]
	N: .02 [1, 2]	[1, 3]	[1, 4]	[1, 5]
		V: .05 [2, 3]	[2, 4]	VP: 1.2e-5 [2, 5]
			Det: .4 [3, 4]	NP: .0012 [3, 5]
				N: .01 [4, 5]

Problem with PCFG

- Lack of sensitivity to structural dependency
- Lack of sensitivity to lexical dependency

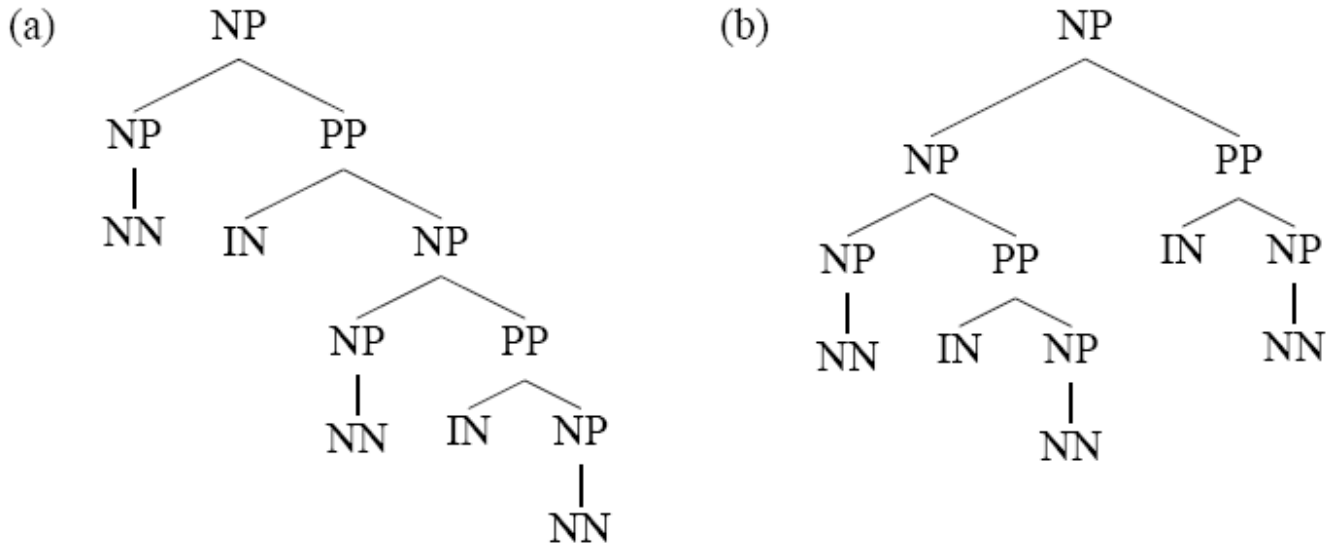
Structure Dependency

- Each PCFG rule is assumed to be independent of each other rule, and thus the rule probabilities are multiplied together.
- Observation: sometimes the choice of how a node expands is dependent on the location of the node in the parse tree
  - NP->Pronoun, NP->Det Noun depends on whether the NP was a subject or an object

	Pronoun	Non-Pronoun
Subject	91%	9%
Object	34%	66%

Structure Dependency

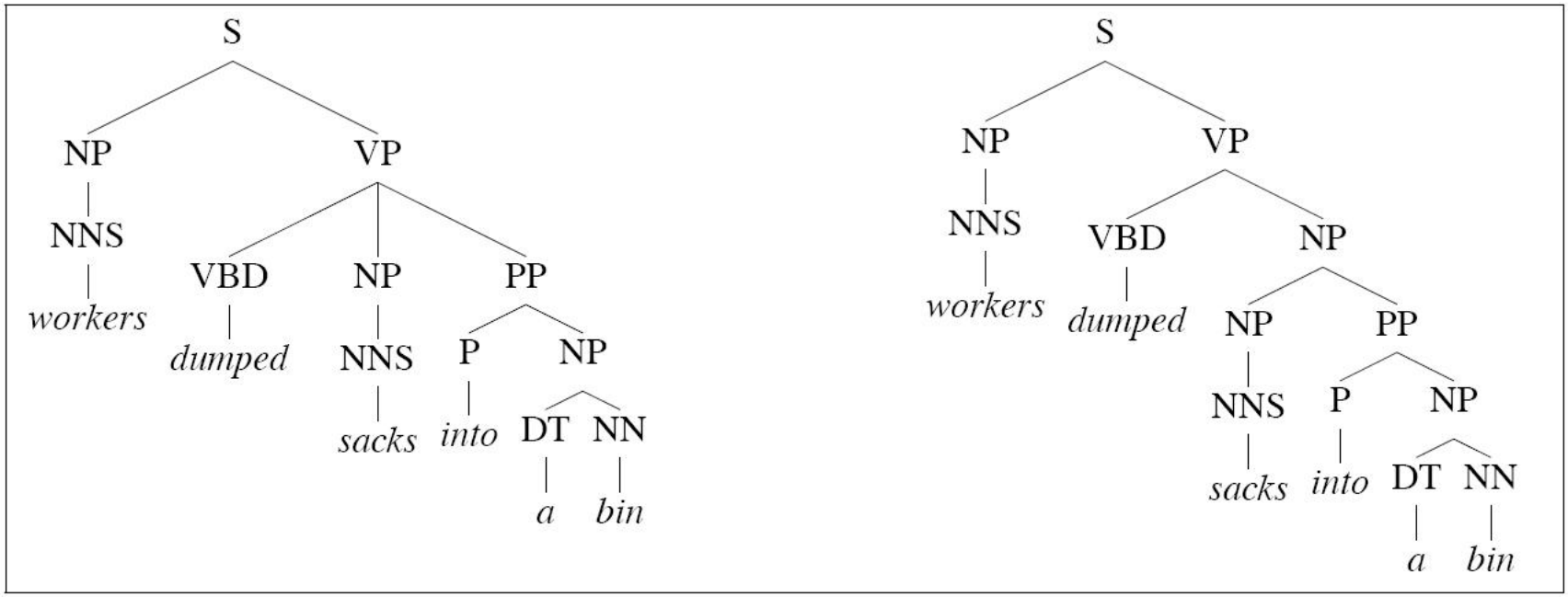
- President of UM in America



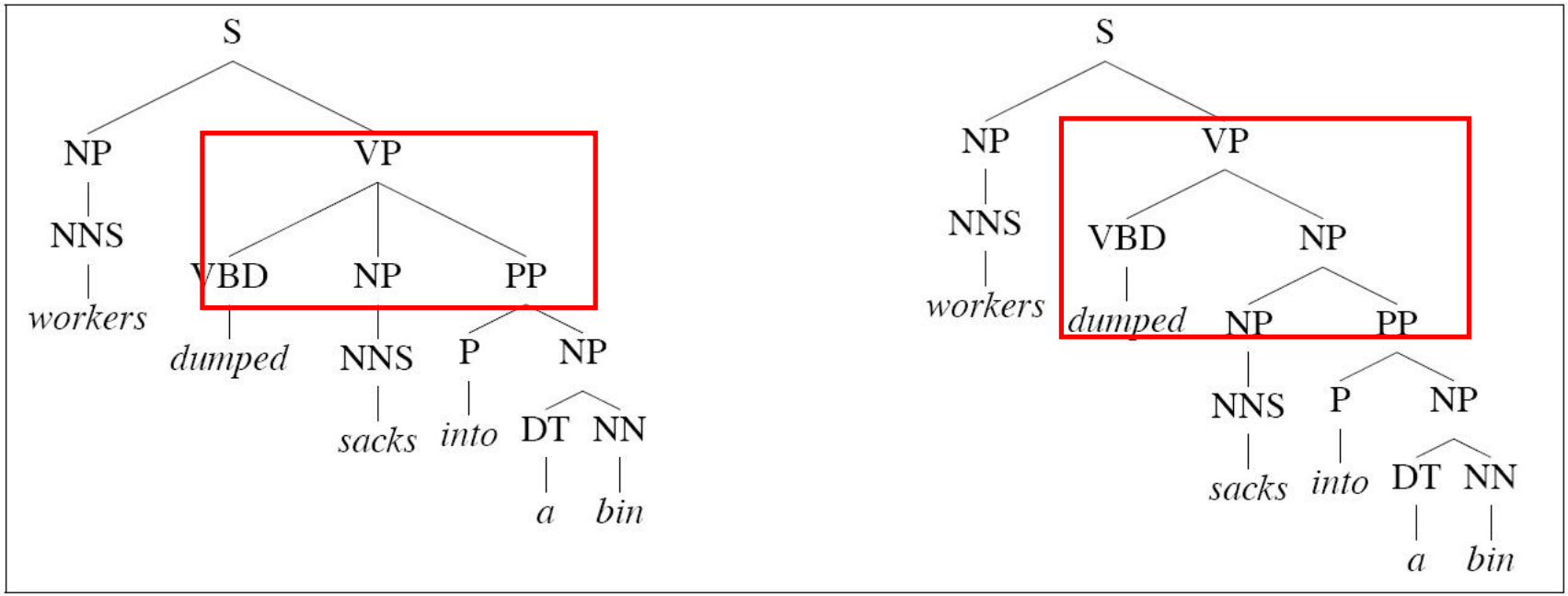
- Both structures have the same rules and therefore receive an equal probability under PCFG.
- However, “close attachment” (structure a) are twice as likely in Wall Street Journal.

Lexical Dependency

- Workers dumped sacks into a bin



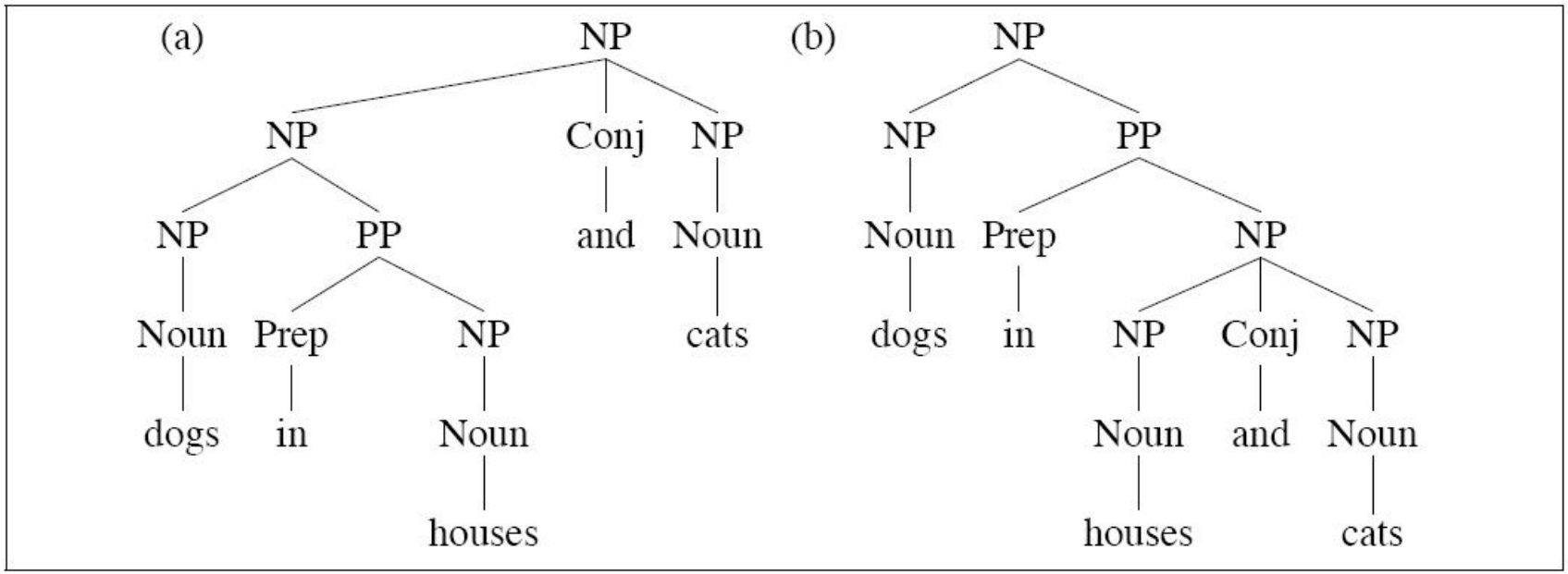
Lexical Dependency



- (a): VP → VBD NP PP
- (b): VP → VBD NP; NP → NP PP
- Depending on these probabilities, a PCFG will always either prefer NP attachment or VP attachment (e.g., NP attachment is slightly more often)

Lexical Dependency

dogs in houses and cats



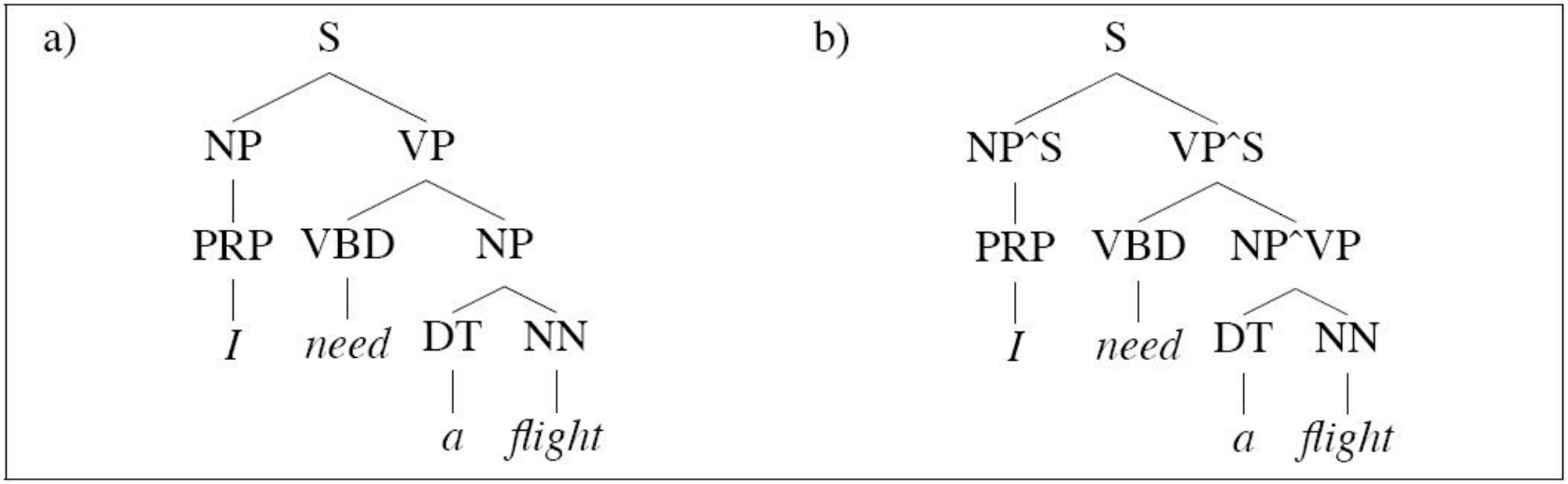
Same probabilities for these two structures

Dealing with Structure Dependency

- How to deal with structure dependency? For example, modeling the fact that NPs in subject position tend to be pronoun and in object position tend to have full lexical form (e.g., Det Nominal).
- Splitting non-terminals
  - E.g., Split NP into two categories: $NP_{subject}$ and NP_{object}
    - $NP_{subject} \rightarrow$Pronoun; $NP_{object} \rightarrow$Pronoun
    - $NP_{subject} \rightarrow$Det Nominal; $NP_{object} \rightarrow$Det Nominal;

Parent Annotation

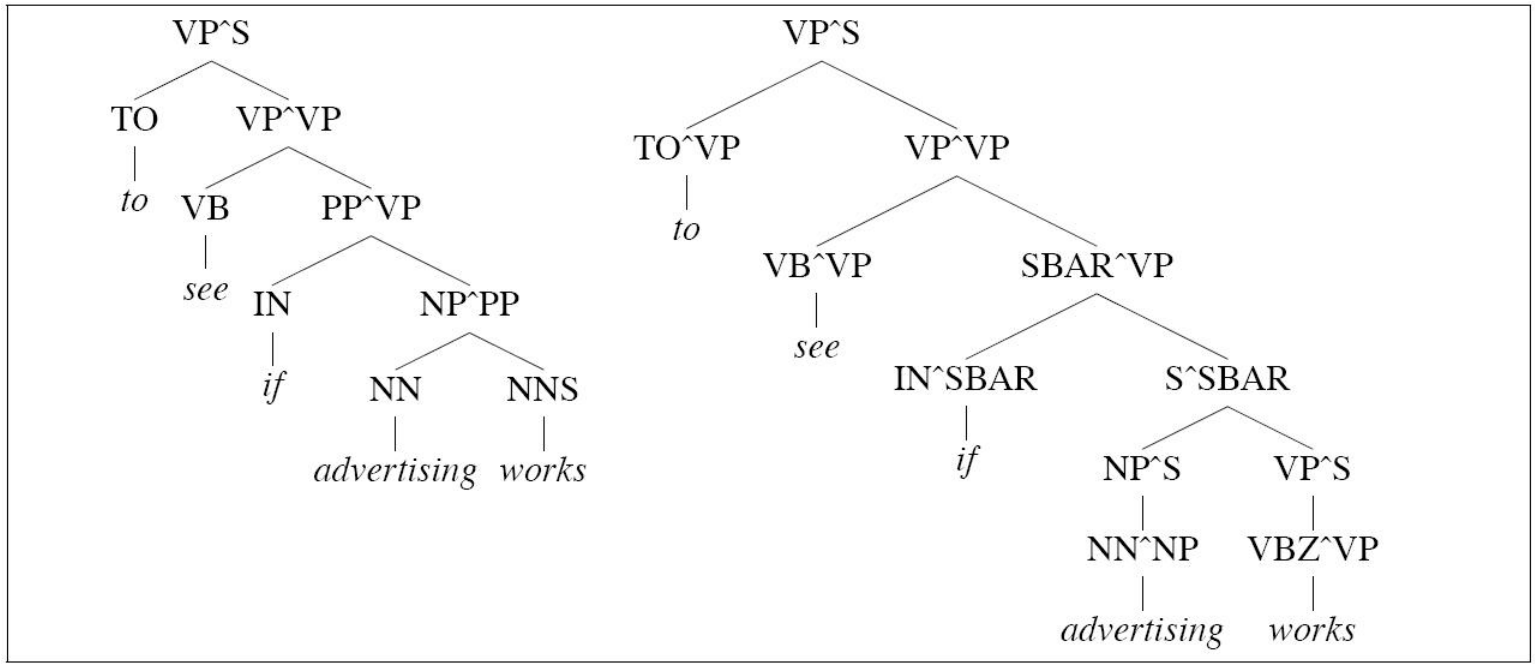
Annotate each phrasal non-terminals with their parents



Previous work also splitting pre-terminal POS nodes (e.g., adverbs)

Splitting Pre-terminals

"IN" can mark a variety of POS including: subordinating conjunctions (while, as, if), complementizers (that, for), and prepositions (of, in, from)



Parent annotation still resulted in an incorrect parse (left)

Splitting pre-terminal nodes resulted in a correct parse (right)

Lexicalized PCFG

- Syntactic constituents could be associated with a lexical **head**.
- Lexicalized grammar can be considered as a simple context free grammar with many copies of each rule.
  - One copy corresponds to each possible headword for each constituent.

VP(dumped) ->VBD(dumped) NP(sacks) PP(into) $3*10^{-10}$

VP(dumped, VBD) ->VBD(dumped, VBD) NP(sacks, NNS) PP(into, IN) : add POS for headword

- Given the lexicalized PCFG, a parser (e.g., Collins) can extend the CKY to do parsing

Heads in CFG

Identify the “head” of each rule

S	⇒	NP	VP
VP	⇒	Vi	
VP	⇒	Vt	NP
VP	⇒	VP	PP
NP	⇒	DT	NN
NP	⇒	NP	PP
PP	⇒	IN	NP

Vi	⇒	sleeps
Vt	⇒	saw
NN	⇒	man
NN	⇒	woman
NN	⇒	telescope
DT	⇒	the
IN	⇒	with
IN	⇒	in

Note: S=sentence, VP=verb phrase, NP=noun phrase, PP=prepositional phrase, DT=determiner, Vi=intransitive verb, Vt=transitive verb, NN=noun, IN=preposition

Rules to Identify Heads

An example for “NP”

If the rule contains NN, NNS, or NNP:
Choose the rightmost NN, NNS, or NNP

Else If the rule contains an NP: Choose the leftmost NP

Else If the rule contains a JJ: Choose the rightmost JJ

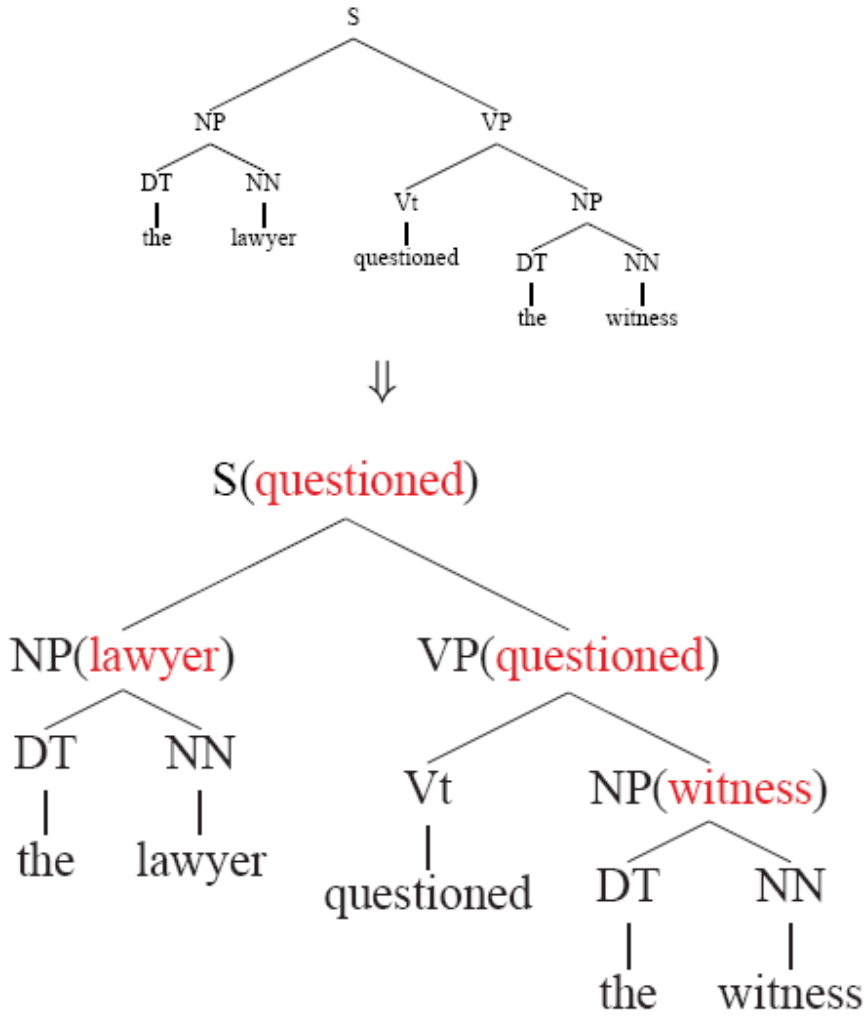
Else If the rule contains a CD: Choose the rightmost CD

Else Choose the rightmost child

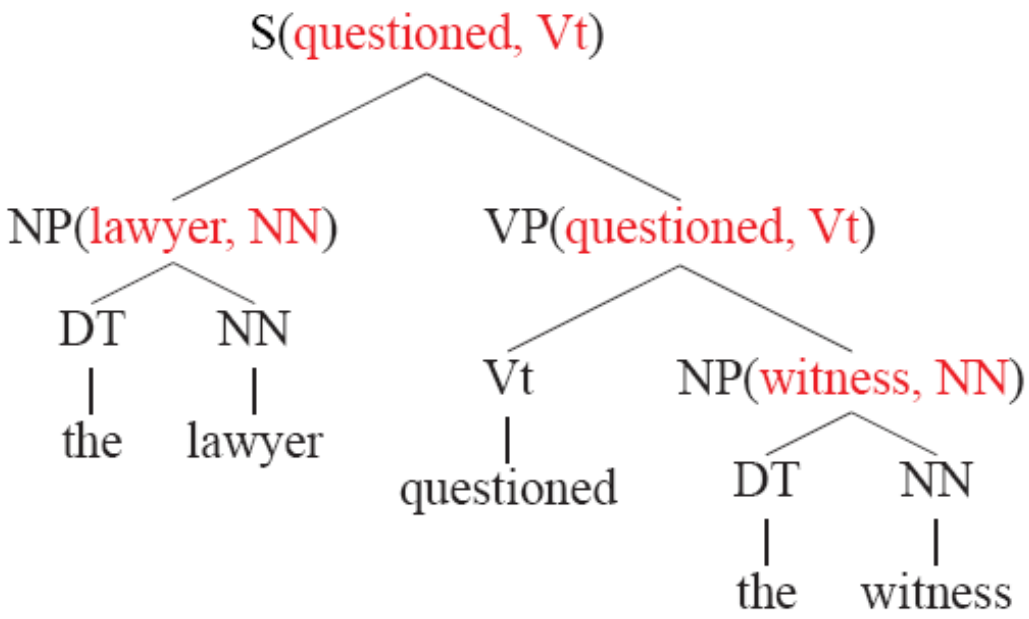
e.g.,

NP	⇒	DT	NNP	NN
NP	⇒	DT	NN	NNP
NP	⇒	NP	PP	
NP	⇒	DT	JJ	
NP	⇒	DT		

Adding Headwords to the Tree

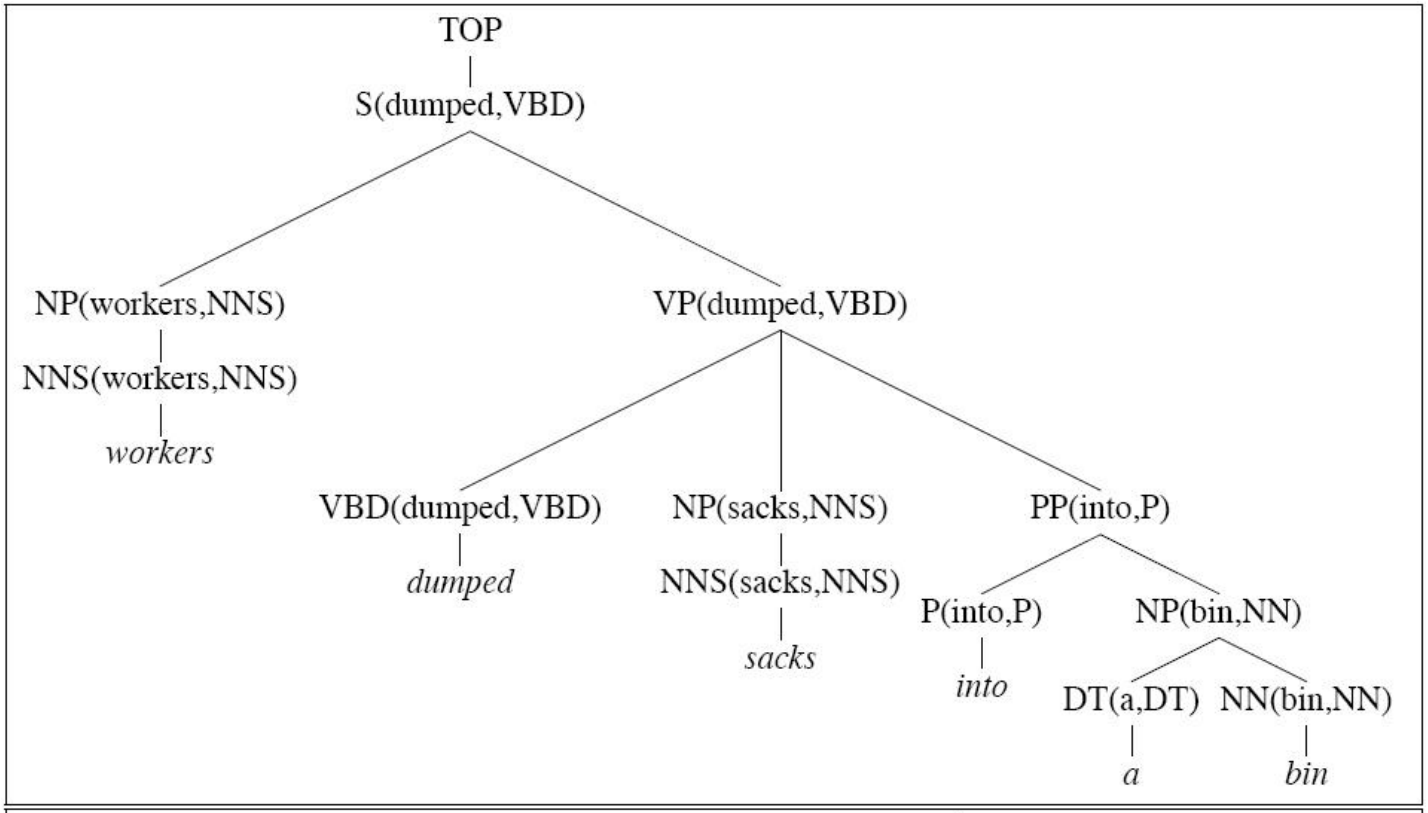


Adding Headwords to the Tree



- A constituent receives its head from its head child.
- The POS tags of the headwords are also propagated up the tree.

An Example



Internal Rules					**Lexical Rules**		
TOP	→	S(dumped,VBD)			NNS(workers,NNS)	→	workers
S(dumped,VBD)	→	NP(workers,NNS)	VP(dumped,VBD)		VBD(dumped,VBD)	→	dumped
NP(workers,NNS)	→	NNS(workers,NNS)			NNS(sacks,NNS)	→	sacks
VP(dumped,VBD)	→	VBD(dumped, VBD)	NP(sacks,NNS)	PP(into,P)	P(into,P)	→	into
PP(into,P)	→	P(into,P)	NP(bin,NN)		DT(a,DT)	→	a
NP(bin,NN)	→	DT(a,DT)	NN(bin,NN)		NN(bin,NN)	→	bin

Key Question

- How to estimate the probabilities for internal rules?

$$\frac{Count(VP(dumped,VBD) \rightarrow VBD(dumped,VBD)NP(sacks,NNS)PP(into,P))}{Count(VP(dumped,VBD))}$$

- Sparse data problem
- Modern statistical parsers differ in which independence assumptions they make.

Anatomy of Lexicalized Rules

- An example lexicalized rule:

VP(dumped, VBD) → VBD(dumped, VBD) NP(sacks, NNS) PP(into, IN)

- Each non-terminal is a triple consisting of:
  - A label, a word, a tag (i.e., part-of-speech tag of the word)

  For example:

  for VP(dumped, VBD): label = VP, word = dumped, tag = VBD

  for VBD(dumped, VBD): label = VBD, word = dumped, tag =VBD

Anatomy of Lexicalized Rules

- An example lexicalized rule:

  VP(dumped, VBD) → VBD(dumped, VBD) NP(sacks, NNS) PP(into, IN)
- The **parent** of the rule is the non-terminal on the left-hand-side (LHS) of the rule
  - In this example: VP(dumped, VBD)
- The **head** of the rule is a single non-terminal on the right-hand-side (RHS) of the rule
  - In this example: VBD(dumped, VBD)
- The **left-modifiers** of the rule are any non-terminal appearing to the left of the head (can be any number 0 or greater)
  - In this example, no left-modifiers
- The **right-modifiers** of the rule are any non-terminals appearing to the right of the head (can be any number 0 or greater)
  - In this example, two right modifiers: NP(sacks, NNS) and PP(into, IN)

The General Form of a Lexicalized Rule

$$X(h,t) \rightarrow L_n(lw_n, lt_n)...L_1(lw_1, lt_1)\, H(h,t)\, R_1(rw_1, rt_1)...R_m(rw_m, rt_m)$$

- *X(h,t)* is the parent of the rule
- *H(h, t)* is the head of the rule
- There are *n* left modifiers, $L_i(lw_i,lt_i)$ for $i = 1..n$
- There are *m* right modifiers, $R_i(rw_i,rt_i)$ for $i = 1..m$
- There can be zero or more left or right modifiers: $n \geq 0$ and $m \geq 0$

Simplified Collins Parser

$$X(h,t) \rightarrow L_n(lw_n, lt_n)...L_1(lw_1, lt_1)\, H(h,t)\, R_1(rw_1, rt_1)...R_m(rw_m, rt_m)$$

VP(dumped, VBD) $\rightarrow$ VBD(dumped, VBD) NP(sacks, NNS) PP(into, IN)

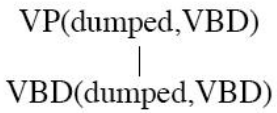
To estimate the probability of the above rule, apply a generative process

- First, given the parent (i.e., LHS X(h,t)), generate the head of the rule
- Then generate the modifiers, one by one, from the inside out.

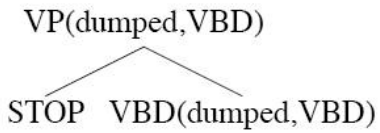
Add STOP at the left and right edges of the rule, which allows the model to know when to stop generating modifiers on a given side.

An Example

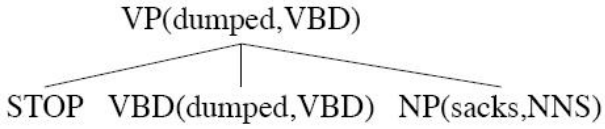
1) Generate the head VBD(dumped,VBD) with probability
P(H|LHS) = P(VBD(dumped,VBD) | VP(dumped,VBD))



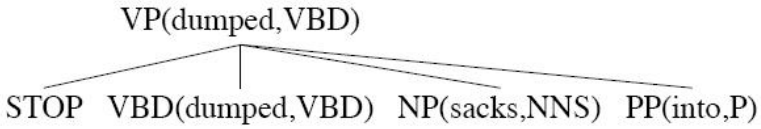
2) Generate the left dependent (which is STOP, since there isn't one) with probability
P(STOP| VP(dumped,VBD) VBD(dumped,VBD))



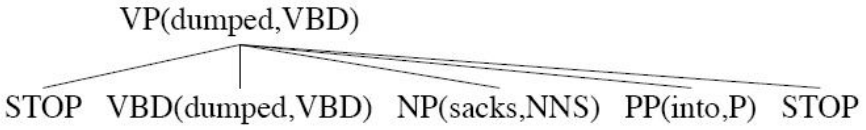
3) Generate right dependent NP(sacks,NNS) with probability
P_r(NP(sacks,NNS)| VP(dumped,VBD), VBD(dumped,VBD))



4) Generate the right dependent PP(into,P) with probability
P_r(PP(into,P) | VP(dumped,VBD), VBD(dumped,VBD))



5) Generate the right dependent STOP with probability
P_r(STOP | VP(dumped,VBD), VBD(dumped,VBD))



The Probability of a Rule

$$P(VP(dumped, VBD) \rightarrow VBD(dumped, VBD)\ NP(sacks, NNS)\ PP(into, P))$$

Can be estimated as the following using conditional independence assumption:

$$\begin{aligned}
&P_H(VBD \mid VP, dumped) \\
&\times P_L(STOP \mid VP, dumped, VBD) \\
&\times P_R(NP(sacks, NNS) \mid VP, dumped, VBD) \\
&\times P_R(PP(in, P) \mid VP, dumped, VBD) \\
&\times P_R(STOP \mid VP, dumped, VBD)
\end{aligned}$$

Parameter Estimation

Use MLE for parameter estimation

$$P_R(NP(sacks, NNS) \mid VP, VBD, dumped)$$
$$= \frac{Count(VP(dumped, VBD)\ with\ \ NNS(sacks, NNS)\ \ as\ \ a\ \ child\ \ somewhere\ \ on\ \ the\ \ right)}{Count(VP(dumped, VBD))}$$

Interpolating backed-off models:

Backoff Level	$P_R(R_i(rw_i, rt_i \mid ...)$	**Example**
1	$P_R(R_i(rw_i, rt_i) \mid P, hw, ht)$	P_R(NP(sacks,NNS)|VP, VBD, dumped)
2	$P_R(R_i(rw_i, rt_i) \mid P, ht)$	$P_R(NP(sacks, NNS) \mid VP, VBD)$
3	$P_R(R_i(rw_i, rt_i) \mid P)$	$P_R(NP(sacks, NNS) \mid VP)$

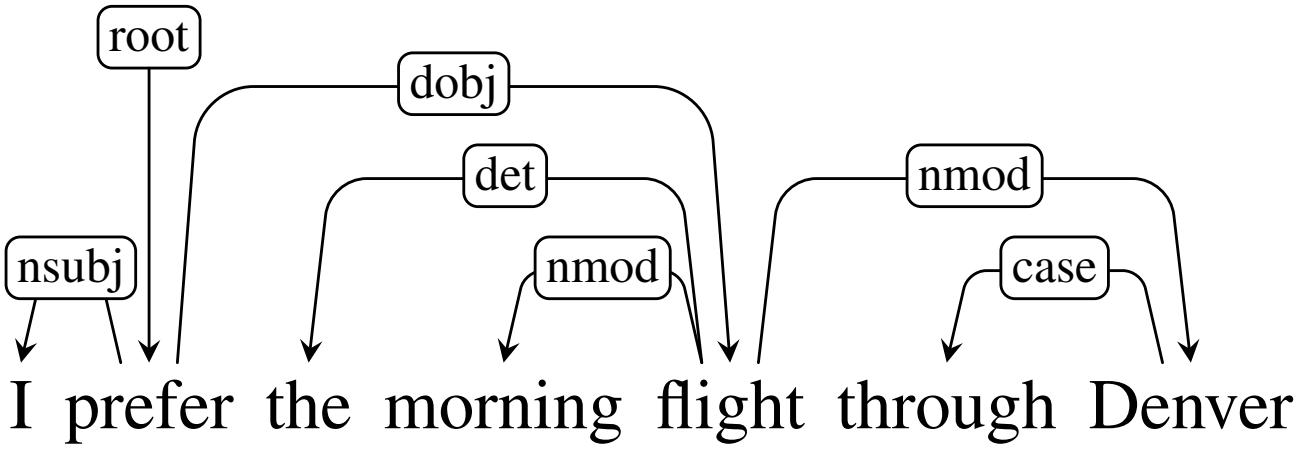
Dependency Parsing

Dependency Grammar

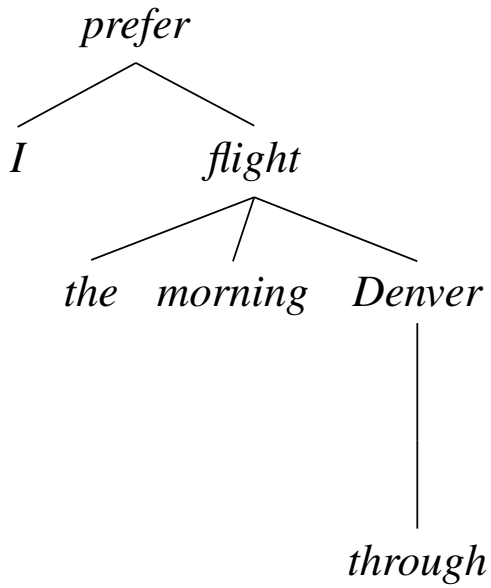
- A completely different formalism from CFG

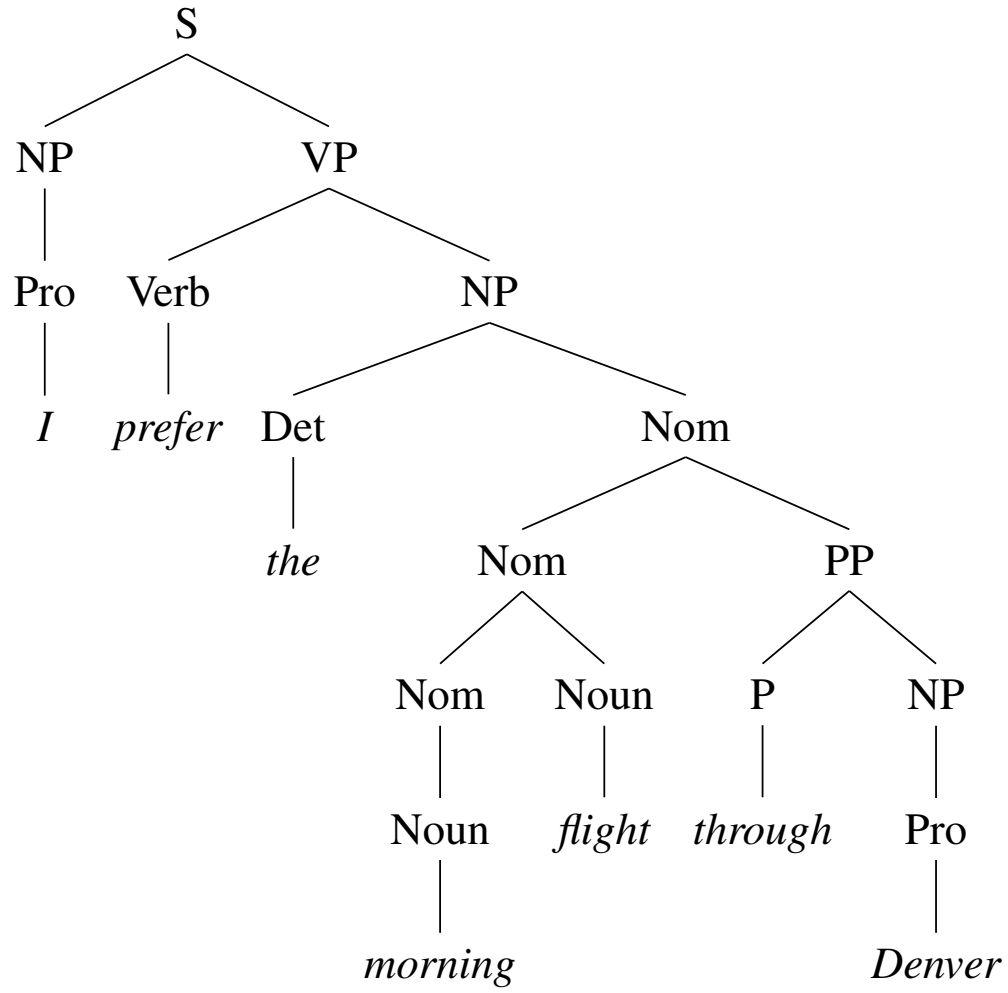
- Syntactic structure of a sentence is described solely in terms of the words (or lemmas) in a sentence and an associated set of directed binary grammatical relations that hold among the words.

Type Dependency Structure



- Relations: directed, labeled arcs from heads to dependents
- Typed relations: from a fixed inventory of grammatical relations
- Root: head of the entire sentence


prefer
I
flight
the
morning
Denver
through


S
NP
VP
Pro
Verb
NP
I
prefer
Det
Nom
the
Nom
PP
Nom
Noun
P
NP
Noun
flight
through
Pro
morning
Denver

Advantages of DG

- Good for languages that have a relatively free word order
  - e.g., Czech, a grammatical object can occur at different position.
- Head-dependent relations provide an approximation to the semantic relationship between predicates and their arguments

Dependency Relations

Clausal Argument Relations	Description
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

Figure 13.2 Selected dependency relations from the Universal Dependency set. (de Marneffe et al., 2014)

The Universal Dependencies project provides an inventory of dependency relations that are linguistically motivated, computationally useful, and cross-linguistically applicable.

Relation	Examples with *head* and **dependent**
NSUBJ	**United** *canceled* the flight.
DOBJ	United *diverted* the **flight** to Reno.
	We *booked* her the first **flight** to Miami.
IOBJ	We *booked* **her** the flight to Miami.
NMOD	We took the **morning** *flight*.
AMOD	Book the **cheapest** *flight*.
NUMMOD	Before the storm JetBlue canceled **1000** *flights*.
APPOS	*United*, a **unit** of UAL, matched the fares.
DET	**The** *flight* was canceled.
	Which *flight* was delayed?
CONJ	We *flew* to Denver and **drove** to Steamboat.
CC	We flew to Denver **and** *drove* to Steamboat.
CASE	Book the flight **through** *Houston*.

Figure 13.3 Examples of core Universal Dependency relations.

Dependency Structure

- *Directed Graph G* = (*V*, *A*) consisting of a set of vertices *V* , and a set of ordered pairs of vertices *A*
- For the most part, the set of vertices, *V* , corresponds exactly to the set of words in a given sentence
- The set of arcs, *A*, captures the head- dependent and grammatical function relationships between the elements in *V*

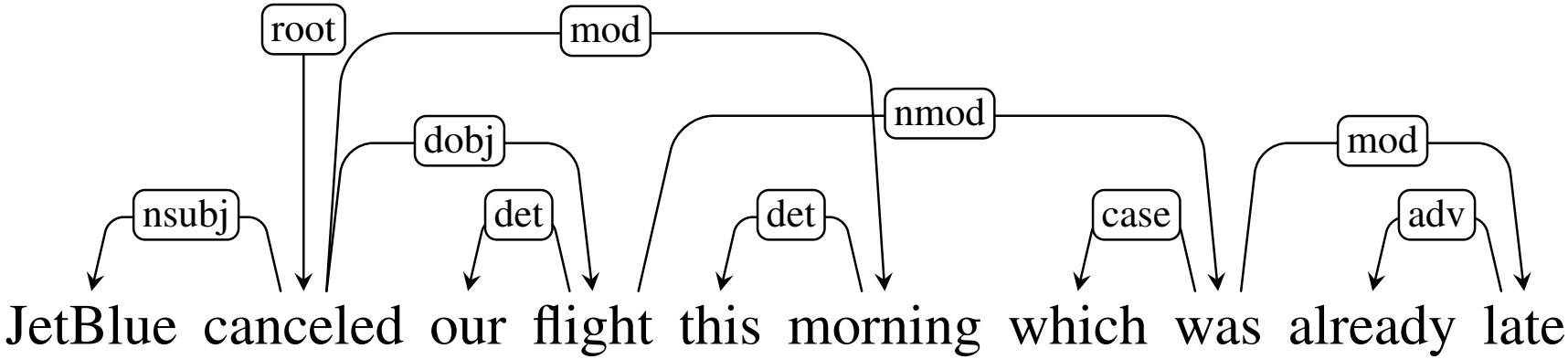
Dependency Tree

- A directed graph that satisfies the following constraints:
  - There is a single designated root node that has no incoming arcs.
  - With the exception of the root node, each vertex has exactly one incoming arc.
  - There is a unique path from the root node to each vertex in V .

Projectivity

- An arc from a head to a dependent is said to be **projective** if there is a path from the head to every word that lies between the head and the dependent in the sentence.
- A dependency tree is then said to be projective if all the arcs that make it up are projective.

Non-Projectivity Example



- Projectivity (and non-projectivity) can be detected in the way we've been drawing our trees.
- A dependency tree is projective if it can be drawn with no crossing edges.

Why Projectivity Matters?

- The most widely used English dependency treebanks were automatically derived from phrase- structure treebanks through the use of head-finding rules -> guaranteed to be projective.

- There are computational limitations to the most widely used families of parsing algorithms. The transition-based approaches can only produce projective trees. This limitation is one of the motivations for the more flexible graph-based parsing approach.

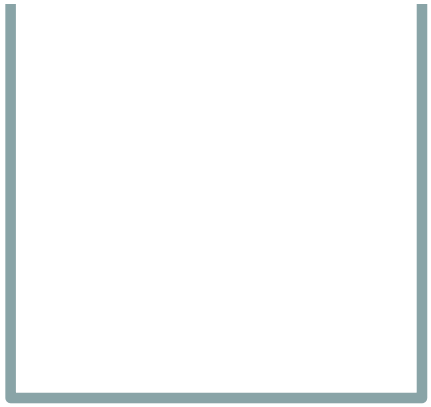
Dependency Treebanks

- Directly annotated dependency treebanks have been created for morphologically rich languages such as Czech, Hindi and Finnish.
- The major English dependency treebanks have largely been extracted from existing resources such as the Wall Street Journal sections of the Penn Treebank, and the more recent OntoNotes projects.
  - Apply rules to identify heads and dependents as well as different grammatical relations
  - Inability to represent non-projective structure

Transition-based Parsing

- Motivated by shift-reduced parsing (Aho and Ullman, 1972)

Stack



Input string:

”abcd”

Grammar (CFG)

A -> bcd
B-> aA

Configuration

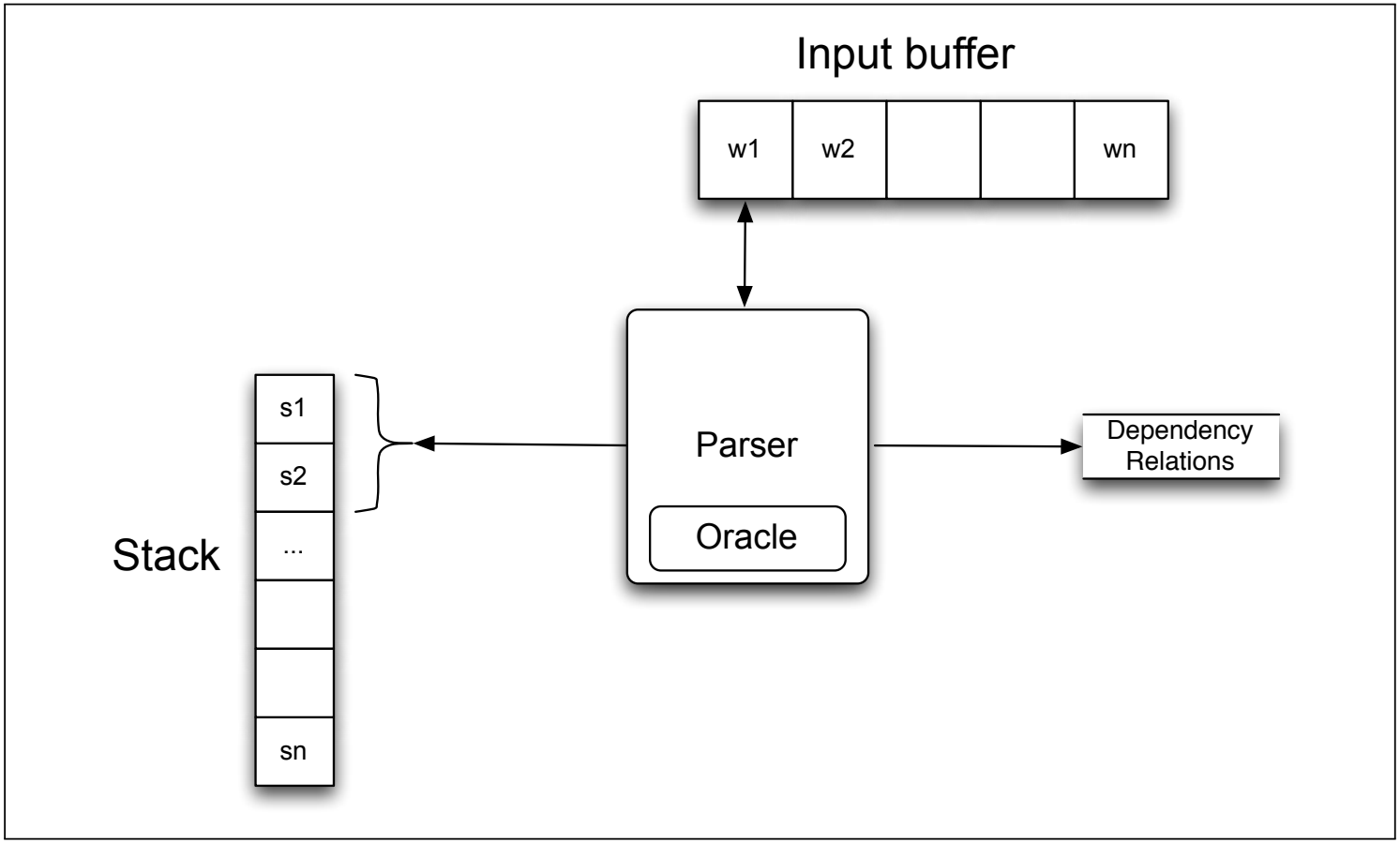
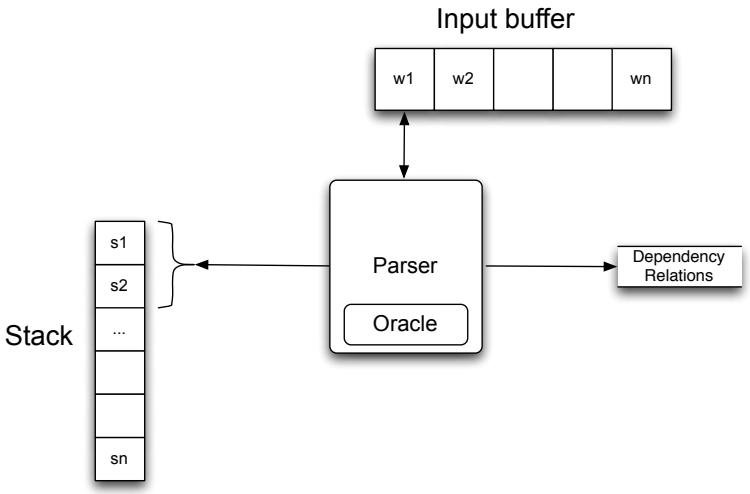


Figure 13.5 Basic transition-based parser. The parser examines the top two elements of the stack and selects an action based on consulting an oracle that examines the current configuration.

Initialization



- Stack contains the ”ROOT” node.
- Buffer contains a sequence of tokens to be parsed
- Empty set of relations

Transition Operators (Arc Standard)

- **LEFTARC:** Assert a head-dependent relation between the word at the top of stack and the word directly beneath it; remove the lower word from the stack.

- **RIGHTARC:** Assert a head-dependent relation between the second word on the stack and the word at the top; remove the word at the top of the stack;

- **SHIFT:** Remove the word from the front of the input buffer and push it onto the stack.

Parsing Algorithm

```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)         ; choose a transition operator to apply
    state ← APPLY(t, state)   ; apply it, creating a new state
  return state
```

Constraints:

- LEFTARC operator cannot be applied when ROOT is the second element of the stack.
- Both reduce operators require two elements to be on the stack to be applied.

Greedy method - the complexity is linear in terms of the length of the sentence.

Trace of Transition Parse

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

Actions are determined by the oracle

- The sequence given is not the only one that might lead to a reasonable parse. In general, there may be more than one path that leads to the same result, and due to ambiguity, there may be other transition sequences that lead to different equally valid parses.

- Given the greedy nature of this algorithm, incorrect choices (based on the oracle) can possibly lead to incorrect parses. The parser has no opportunity to go back and pursue alternative choices.

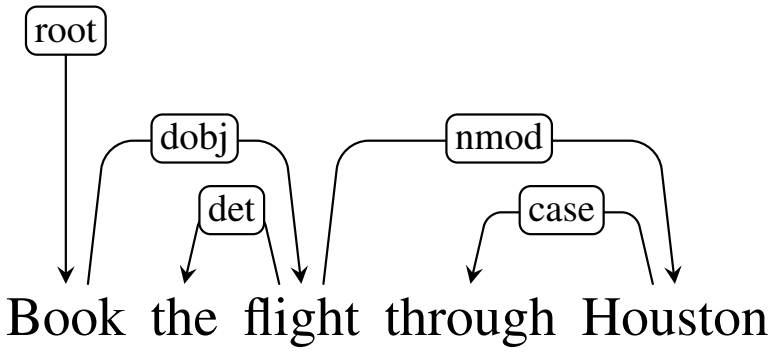
- In practice operators are parameterized with dependency labels: LEFTARC(NSUBJ) or RIGHTARC(DOBJ) -> make the job of oracle more difficult since it has a much larger space to choose from

Creating an Oracle

- Use supervised machine learning methods to train classifiers that play the role of the oracle. Given appropriate training data, these methods learn a function that maps from configurations to transition operators.

- Given a reference parse and a configuration, the training oracle proceeds as follows:
  - Choose LEFTARC if it produces a correct head-dependent relation given the reference parse and the current configuration,
  - Otherwise, choose RIGHTARC if (1) it produces a correct head-dependent relation given the reference parse and (2) all of the dependents of the word at the top of the stack have already been assigned,
  - Otherwise, choose SHIFT.

Creating training data



Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Train a classifier

- Given configuration-transition pairs, extract useful features from the configurations to train classifiers.

- Features from the configuration (stack, buffer, current set of relations):

Source	Feature templates		
One word	$s_1.w$	$s_1.t$	$s_1.wt$
	$s_2.w$	$s_2.t$	$s_2.wt$
	$b_1.w$	$b_1.w$	$b_0.wt$
Two word	$s_1.w \circ s_2.w$	$s_1.t \circ s_2.t$	$s_1.t \circ b_1.w$
	$s_1.t \circ s_2.wt$	$s_1.w \circ s_2.w \circ s_2.t$	$s_1.w \circ s_1.t \circ s_2.t$
	$s_1.w \circ s_1.t \circ s_2.t$	$s_1.w \circ s_1.t$	

Stack	Word buffer	Relations
[root, canceled, flights]	[to Houston]	(canceled → United) (flights → morning) (flights → the)

$$\langle s_1.w = flights, op = shift \rangle$$
$$\langle s_2.w = canceled, op = shift \rangle$$
$$\langle s_1.t = NNS, op = shift \rangle$$
$$\langle s_2.t = VBD, op = shift \rangle$$
$$\langle b_1.w = to, op = shift \rangle$$
$$\langle b_1.t = TO, op = shift \rangle$$
$$\langle s_1.wt = flightsNNS, op = shift \rangle$$

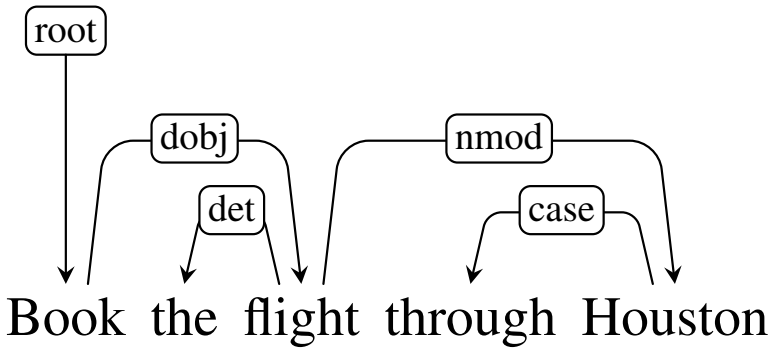
$$\langle s_1.t \circ s_2.t = NNSVBD, op = shift \rangle$$

Learning

- Over the years, the dominant approaches to training transition-based dependency parsers have been multinomial logistic regression and support vector machines, both of which can make effective use of large numbers of sparse features.

- More recently, deep learning approaches have been applied successfully to transition-based parsing. These approaches eliminate the need for complex, hand-crafted features and have been particularly effective at overcoming the data sparsity issues normally associated with training transition-based parsers.

Arc-standard approach delays the head assignment



Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Arc Eager Approach

- **LEFTARC:** Assert a head-dependent relation between the word at the front of the input buffer and the word at the top of the stack; pop the stack.
- **RIGHTARC**: Assert a head-dependent relation between the word on the top of the stack and the word at front of the input buffer; shift the word at the front of the input buffer to the stack.
- **SHIFT:** Remove the word from the front of the input buffer and push it onto the stack.
- **REDUCE:** Pop the stack.

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, the, flight, through, houston]	RIGHTARC	(root → book)
1	[root, book]	[the, flight, through, houston]	SHIFT	
2	[root, book, the]	[flight, through, houston]	LEFTARC	(the ← flight)
3	[root, book]	[flight, through, houston]	RIGHTARC	(book → flight)
4	[root, book, flight]	[through, houston]	SHIFT	
5	[root, book, flight, through]	[houston]	LEFTARC	(through ← houston)
6	[root, book, flight]	[houston]	RIGHTARC	(flight → houston)
7	[root, book, flight, houston]	[]	REDUCE	
8	[root, book, flight]	[]	REDUCE	
9	[root, book]	[]	REDUCE	
10	[root]	[]	Done	

The arc-eager approach permits a word to be eagerly assigned its head and still allow it to serve as the head for later dependents.

Graph-based Dependency Parsing

- Search through the space of possible trees for a given sentence for a tree (or trees) that maximize some score.
- Given a sentence *S* we're looking for the best dependency tree in $\mathcal{G}_S$, the space of all possible trees for that sentence, that maximizes:

$$\hat{T}(S) = \underset{t \in \mathcal{G}_S}{\operatorname{argmax}} \, score(t, S)$$

- Score for a tree is based on the scores of the edges that comprise the tree.

$$score(t, S) = \sum_{e \in t} score(e)$$

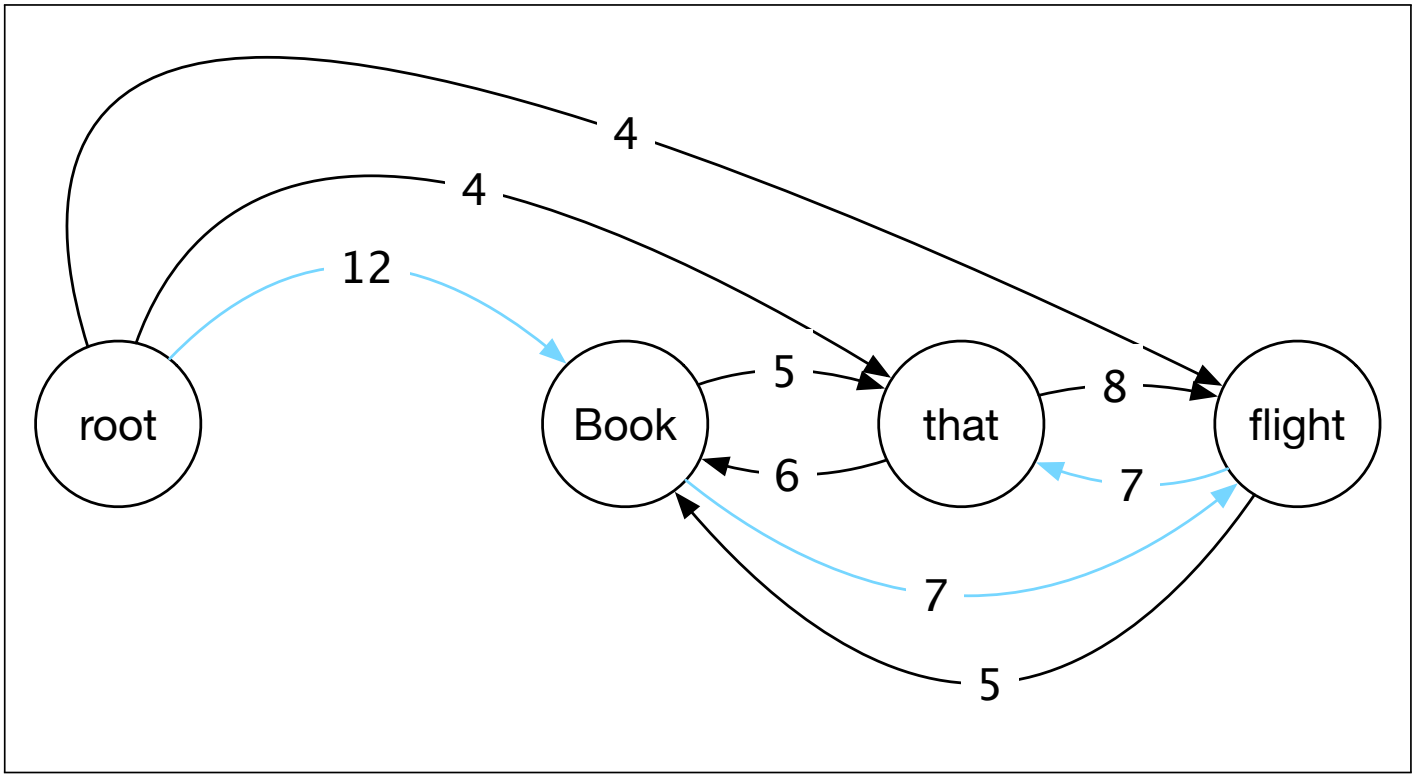
Motivation for Graph-based Parsing

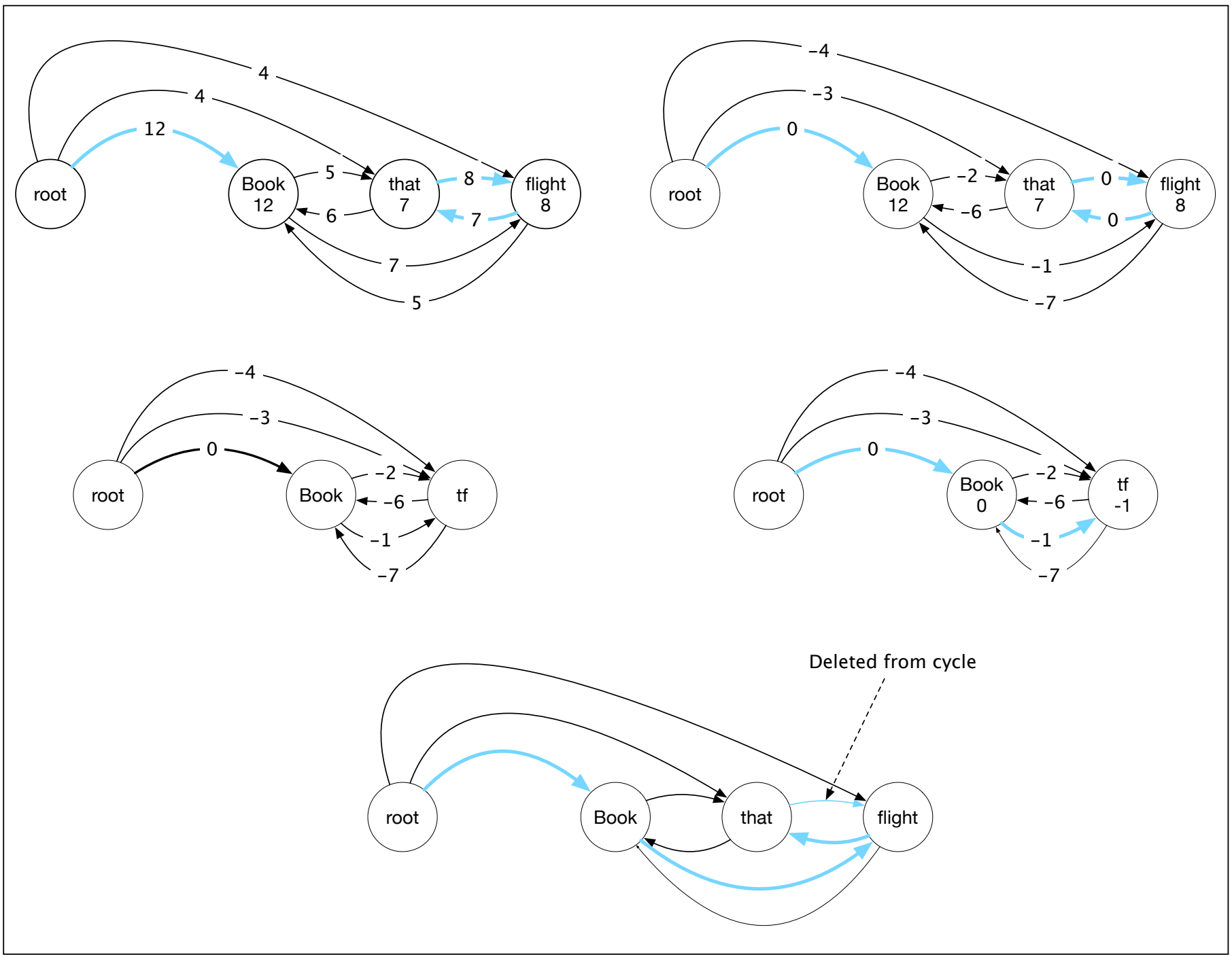
- Unlike transition-based approaches, these methods are capable of producing non-projective trees.
- Transition-based methods have high accuracy on shorter dependency relations but accuracy declines significantly as the distance between the head and dependent increases.
  - Graph-based methods avoid this difficulty by scoring entire trees, rather than relying on greedy local decisions

Graph-based Parsing Algorithm

- Given an input sentence, construct a fully-connected, weighted, directed graph where the vertices are the input words and the directed edges represent *all possible* head-dependent assignments.
- A maximum spanning tree of this graph emanating from the ROOT represents the preferred dependency parse for the sentence.
  - Greedy selection process: for each vertex, choose the incoming edge with highest score.
  - Recursively clean up and eliminate cycles.

”book that flight”: maximum spanning tree corresponding to the desired parse shown in blue

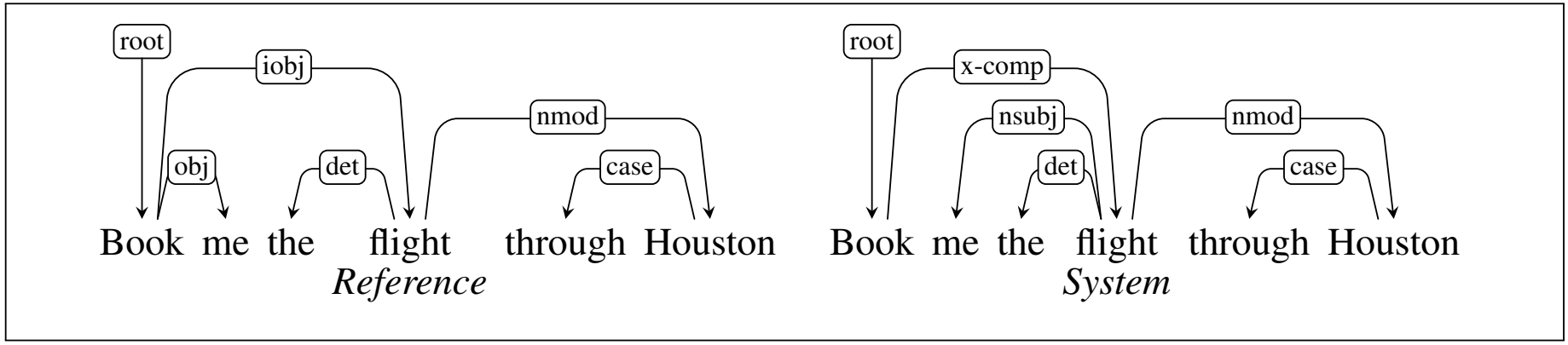



root
Book
12
that
7
flight
8
4
4
12
5
6
8
7
7
5
root
Book
12
that
7
flight
8
−4
−3
0
−2
−6
0
0
−1
−7
root
Book
tf
−4
−3
0
−2
−6
−1
−7
root
Book
0
tf
-1
−4
−3
0
−2
−6
−1
−7
Deleted from cycle
root
Book
that
flight

Evaluation

Labeled and unlabeled attachment accuracy

- **Labeled attachment** refers to the proper assignment of a word to its head along with the correct dependency relation.
- **Unlabeled attachment** simply looks at the correctness of the assigned head, ignoring the dependency relation.



LAS: 4/6; UAS: 5/6