8 - Context Free Grammars and Syntactic Parsing

CS 6320

Outline

- Formal Grammars
  - Context-free grammar
  - Grammars for English
  - Treebanks
  - Dependency grammars

- Syntactic Parsing
  - Bottom-up, top-down
  - Ambiguity
  - CKY parsing
  - Earley parsing

Syntax

- Syntax: provides rules to put together words to form components of sentence and to put together these components to form sentences.
- Knowledge of syntax is useful for:
  - Parsing
  - QA
  - IE
  - Generation
  - Translation, etc.
- Grammar is the formal specification of rules of a language.
- Syntactic Parsing is a method to perform syntactic analysis of a sentence.

Syntax

- Key notions that we'll cover
  - Constituency
  - Grammatical relations and Dependency
    - Heads
- Key formalism
  - Context-free grammars
- Resources
  - Treebanks

Constituency

- The basic idea here is that groups of words within utterances can be shown to act as single units.
- And in a given language, these units form coherent classes that can be shown to behave in similar ways
  - With respect to their internal structure
  - And with respect to other units in the language

Constituency

- Internal structure
  - We can describe an internal structure to the class (might have to use disjunctions of somewhat unlike sub-classes to do this).
- External behavior
  - For example, we can say that noun phrases can come before verbs

Constituency

- For example, it makes sense to say that the following are all *noun phrases* in English...

Harry the Horse	a high-class spot such as Mindy's
the Broadway coppers	the reason he comes into the Hot Box
they	three parties from Brooklyn

- Why? One piece of evidence is that they can all precede verbs.
  - This is external evidence

Grammars and Constituency

- Of course, there's nothing easy or obvious about how we come up with right set of constituents and the rules that govern how they combine...
- That's why there are so many different theories of grammar and competing analyses of the same data.
- The approach to grammar, and the analyses, adopted here are very generic (and don't correspond to any modern linguistic theory of grammar).

Chomsky's Classification 1/2

- Chomsky identifies four classes of grammars:
  - **Class 0**: unrestricted phrase-structure grammars. No restriction on type of rules. (Turing equivalent)

$$x \rightarrow y$$

  - **Class 1**: context sensitive grammars.

$$xAy \rightarrow xzy$$

    Rewrite a non-terminal *A* in context *xAy*

  - **Class 2**: Context free grammars

$$A \rightarrow x$$

    A is a nonterminal.

    x is a sequence of terminals and/or nonterminal symbols.

Chomsky's Classification 2/2

- **Class 3**: regular grammars

$$A \rightarrow Bt$$

$$or$$

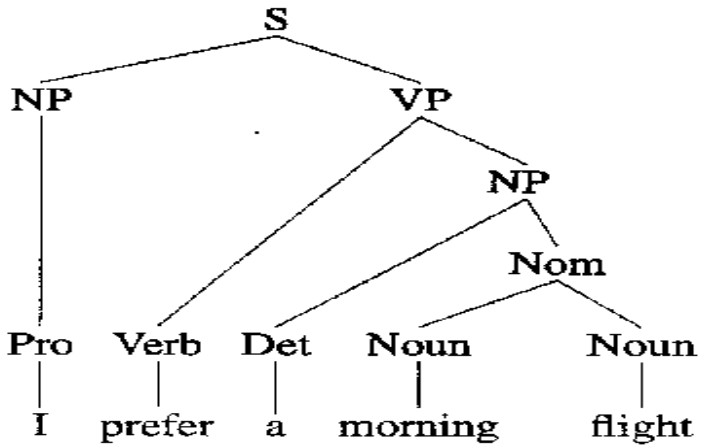
$$A \rightarrow t$$

A, B are nonterminals

t is a terminal.

Context Free Grammars

- Just as with FSAs, one can view the grammar rules as either structure imposing device or generative device.
- A derivation is a sequence of rule applications.
- Derivations can be visualized as parse trees.



- Compare CFG with:
  - Regular expressions (too weak)
  - Context sensitive grammars (too strong)
  - Turing machines (way too strong)

Context-Free Grammars

- Context-free grammars (CFGs)
  - Also known as
    - Phrase structure grammars
    - Backus-Naur form
- Consist of
  - Rules
  - Terminals
  - Non-terminals

Context-Free Grammars

- Terminals
  - We'll take these to be words (for now)
- Non-Terminals
  - The constituents in a language
    - Like noun phrase, verb phrase and sentence
- Rules
  - Rules are equations that consist of a single non-terminal on the left and any number of terminals and non-terminals on the right.

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

Definition

- More formally, a CFG consists of

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 $A \rightarrow \beta$,
where A is a non-terminal,
β is a string of symbols from the infinite set of strings $(\Sigma \cup N)*$

S a designated **start symbol**

L0 Grammar

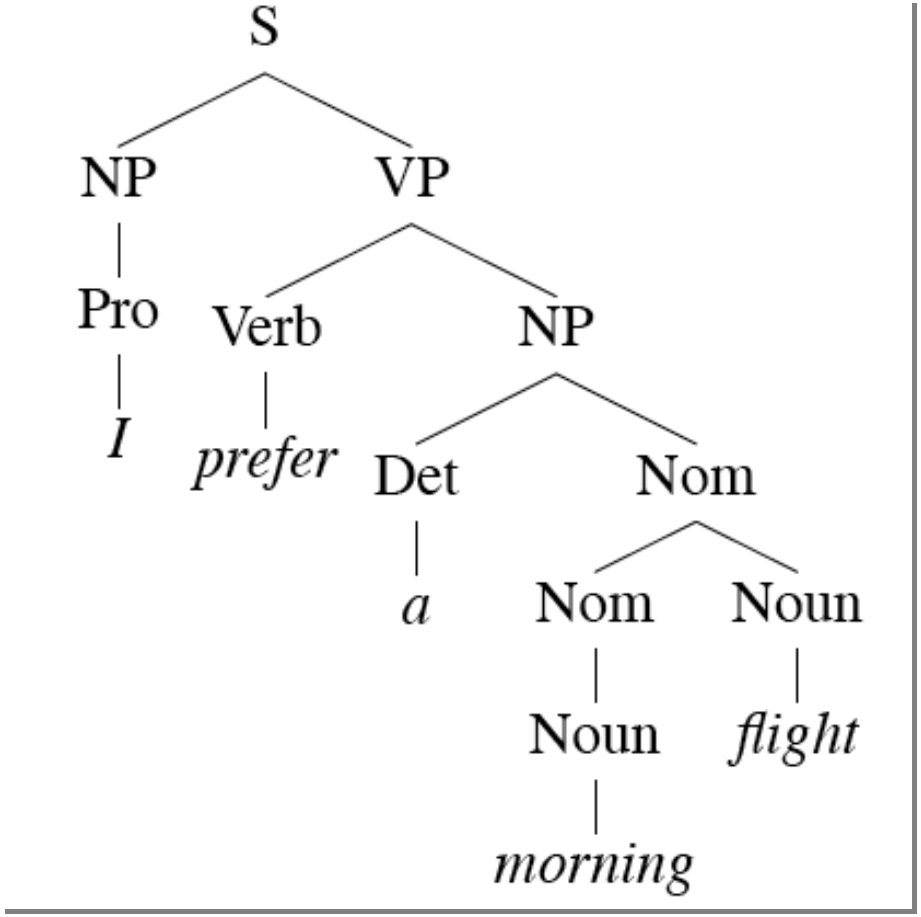
Grammar Rules			Examples
S	→	*NP VP*	I + want a morning flight
NP	→	*Pronoun*	I
	|	*Proper-Noun*	Los Angeles
	|	*Det Nominal*	a + flight
Nominal	→	*Nominal Noun*	morning + flight
	|	*Noun*	flights
VP	→	*Verb*	do
	|	*Verb NP*	want + a flight
	|	*Verb NP PP*	leave + Boston + in the morning
	|	*Verb PP*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

Generativity

- As with FSAs and FSTs, 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



Parsing

- Parsing is the process of taking a string and a grammar and returning a (multiple?) parse tree(s) for that string
- It is completely analogous to running a finite-state transducer with a tape
  - It's just more powerful
    - Remember this means that there are languages we can capture with CFGs that we can't capture with finite-state methods

An English Grammar Fragment

- Sentences
- Noun phrases
  - Agreement
- Verb phrases
  - Subcategorization

Sentence Types

- Declaratives: *A plane left.*

  $S \rightarrow NP\ VP$

- Imperatives: *Leave!*

  $S \rightarrow VP$

- Yes-No Questions: *Did the plane leave?*

  $S \rightarrow Aux\ NP\ VP$

- WH Questions: *When did the plane leave?*

  $S \rightarrow WH\text{-}NP\ Aux\ NP\ VP$

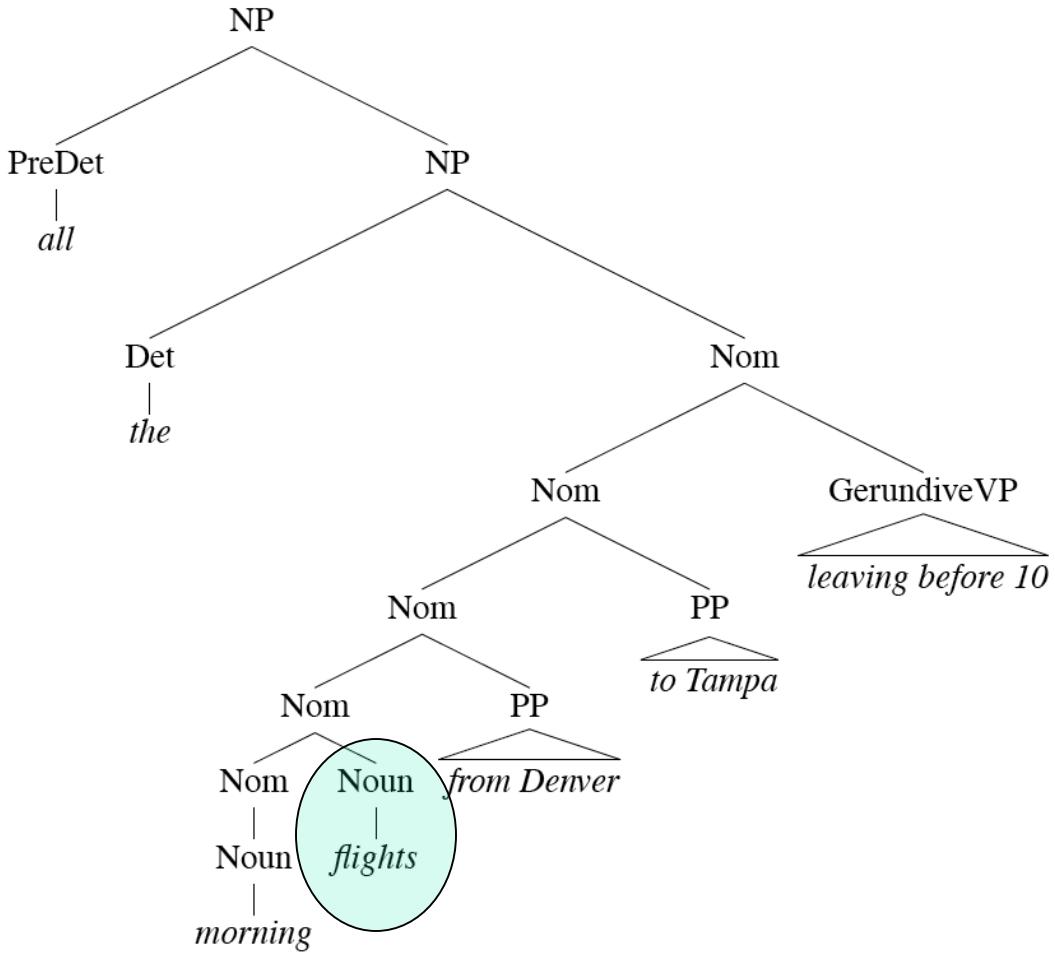
Noun Phrases

- 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 criticial noun in this NP. Let's call that the *head*.
- We can dissect this kind of NP into the stuff that can come before the head, and the stuff that can come 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
      - large cars
    - Ordering constraints
      - Three large cars
      - ?large three cars

Postmodifiers

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

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	*This flights
Those flights	*Those flight

$$Does[_{NP} this\ \ flight]\ stop\ in\ Dallas?$$

$$S \rightarrow Aux\ NP\ VP$$

Such rules need to have agreements in number, gender, case.

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.*
  - We'll come back to this in a bit

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

Some Difficulties

- Subcategorization:
  - Verbs have preference for the kind of constituents they cooccur with. Not every verb is compatible with every verb phrase.

    Example: want can be used with NP complement, or VP complement.

    I want a flight.

    I want to fly.

    But not other verbs:

    *I found to fly…

Subcategorization

- We say that *find* subcategorizes for an NP while *want* subcategorizes for NP or a nonfinite VP.
- Complements, are called subcategorization frames.

Frame	Verb	Example
$\emptyset$	eat, sleep	I want to eat
NP	prefer, find, leave,	Find [$_{NP}$ the flight from Pittsburgh to Boston]
NP NP	show, give	Show [$_{NP}$ me] [$_{NP}$ airlines with flights from Pittsburgh]
PP_{from} PP_{to}	fly, travel	I would like to fly [$_{PP}$ from Boston] [$_{PP}$ to Philadelphia]
NP PP_{with}	help, load,	Can you help [$_{NP}$ me] [$_{PP}$ with a flight]
VPto	prefer, want, need	I would prefer [$_{VPto}$ to go by United airlines]
VPbrst	can, would, might	I can [$_{VPbrst}$ go from Boston]
S	mean	Does this mean [$_{S}$ AA has a hub in Boston]?

- Movement:

  I looked up his grade.

  I looked his grade up.

Subcategorization

- Even though there are many valid VP rules in English, not all verbs are allowed to participate in all those VP rules.
- We can subcategorize the verbs in a language according to the sets of VP rules that they participate in.
- This is a modern take on the traditional notion of transitive/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

Why?

- 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

Recursive Structures

- Recursive rules: one rules where the nonterminal on the left-hand side also appears on the righthand side.

$$NP \rightarrow NP\ PP \quad The\ flight\ from\ Boston$$
$$VP \rightarrow VP\ PP \quad departed\ Miami\ at\ noon.$$

- This allows us to do 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
  - Flights to Miami from Boston in April on Friday under $300 with lunch

Conjunctions

$$S \rightarrow S \; and \quad S$$

$$NP \rightarrow NP \; and \quad NP$$

$$VP \rightarrow VP \; and \quad VP$$

- Any phrasal constituent can be conjoined with a constituent of the same type to form a new constituent of that type. We can say that English has the rule:

$$X \rightarrow X \; and \; X$$

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.

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

```
( (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

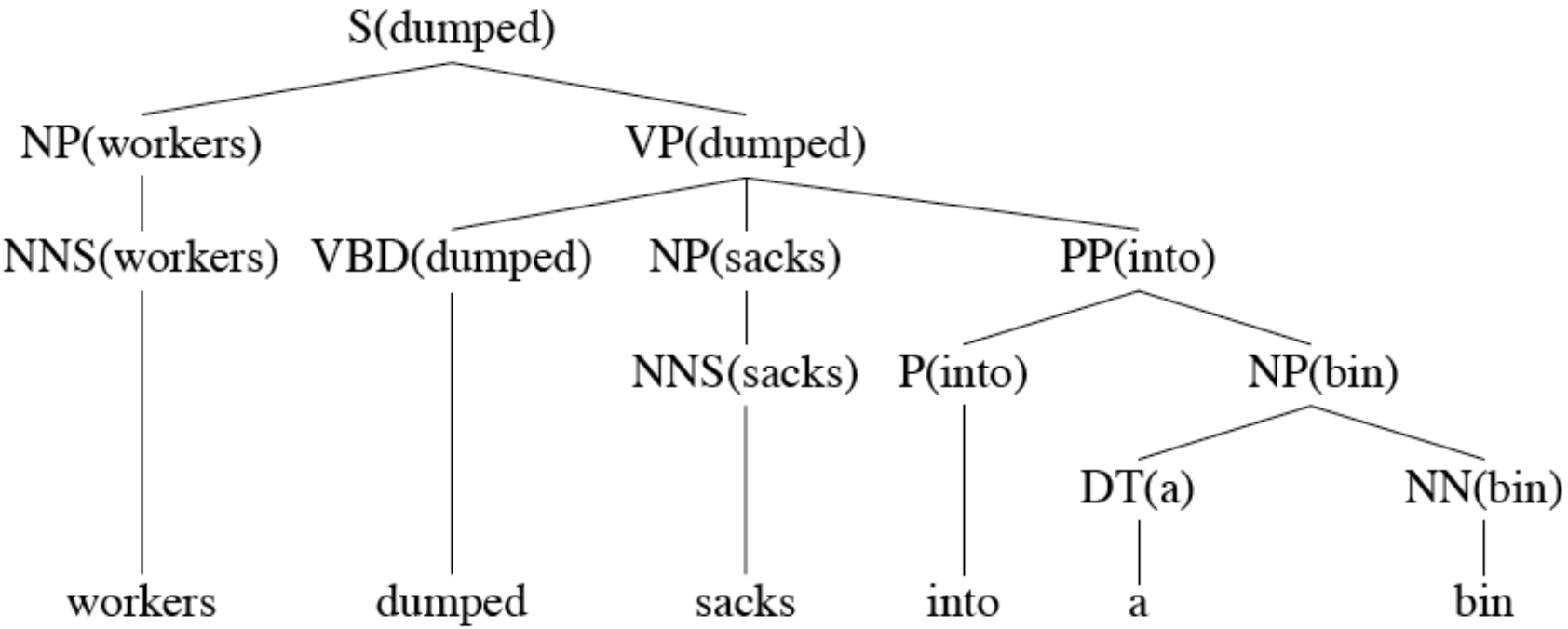
- Such grammars tend to be very flat due to the fact that they tend to avoid recursion.
  - To ease the annotators burden
- For example, the Penn Treebank has 4500 different rules for VPs. Among them...

```
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.

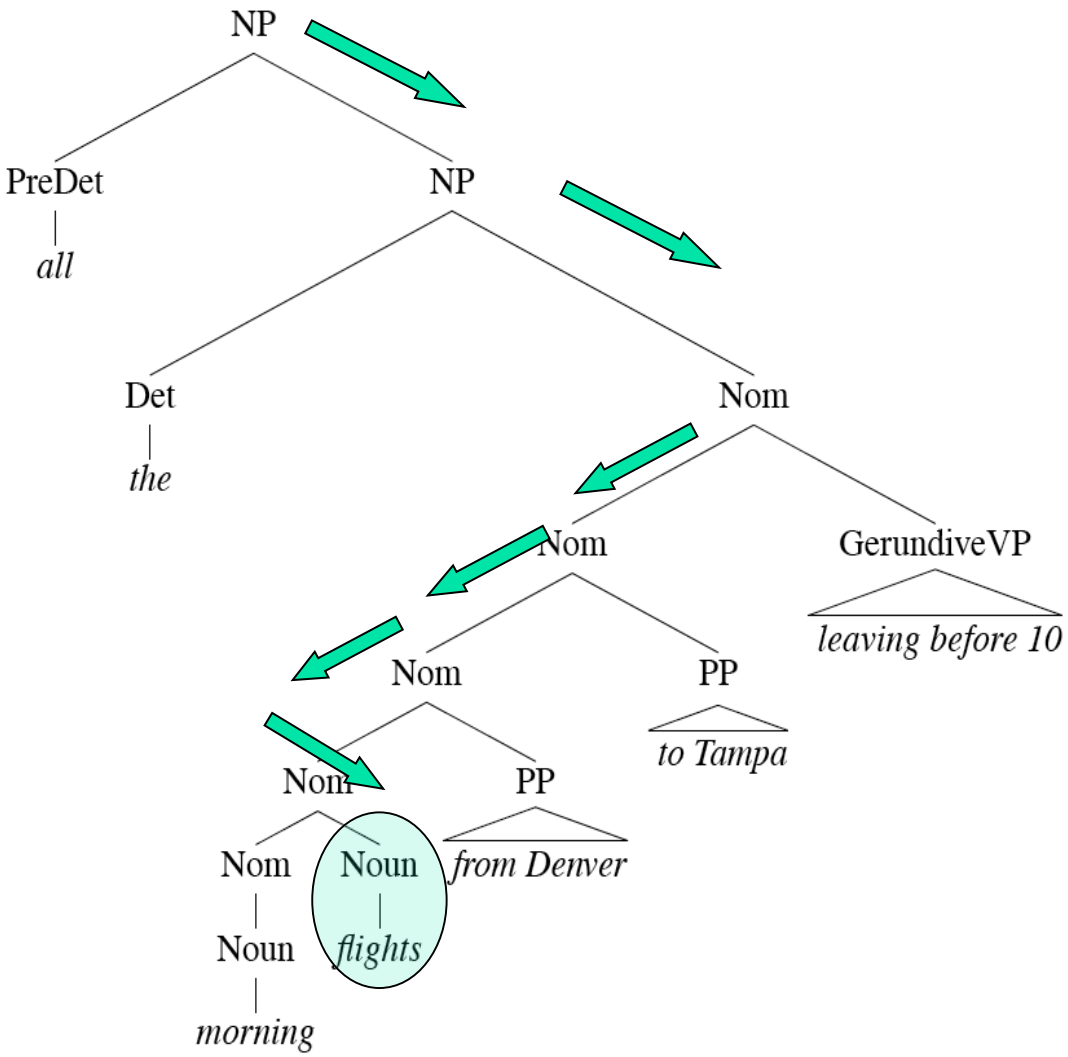
Lexically Decorated Tree



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.

Noun Phrases



Treebank Uses

- Treebanks (and headfinding) are particularly critical to the development of statistical parsers
  - Chapter 14
- Also valuable to *Corpus Linguistics*
  - Investigating the empirical details of various constructions in a given language

Summary of CFG

- 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

- 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

Parsing

- As with everything of interest, parsing involves a search which involves the making of choices
- We'll start with some basic (meaning bad) methods before moving on to the one or two that you need to know

For Now

- Assume…
  - You have all the words already in some buffer
  - The input isn't POS tagged
  - We won't worry about morphological analysis
  - All the words are known

  - These are all problematic in various ways, and would have to be addressed in real applications.

A Simple Grammar

$S \rightarrow NP\ VP$

$VP \rightarrow V\ NP$

$NP \rightarrow NAME$

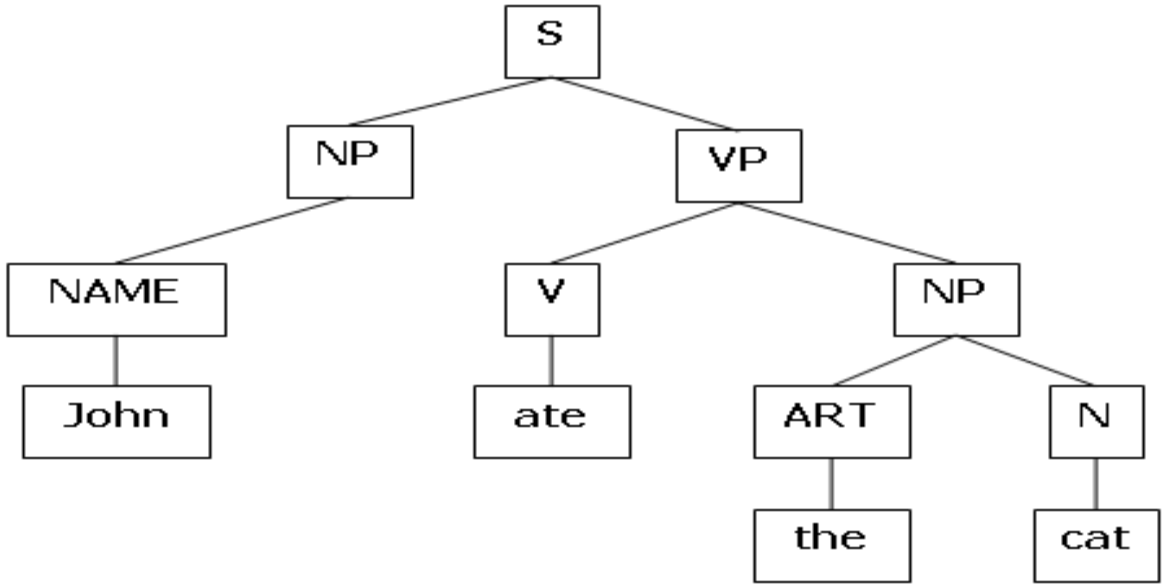
$NP \rightarrow ART\ N$

$NAME \rightarrow John$

$V \rightarrow ate$

$ART \rightarrow the$

$N \rightarrow cat$

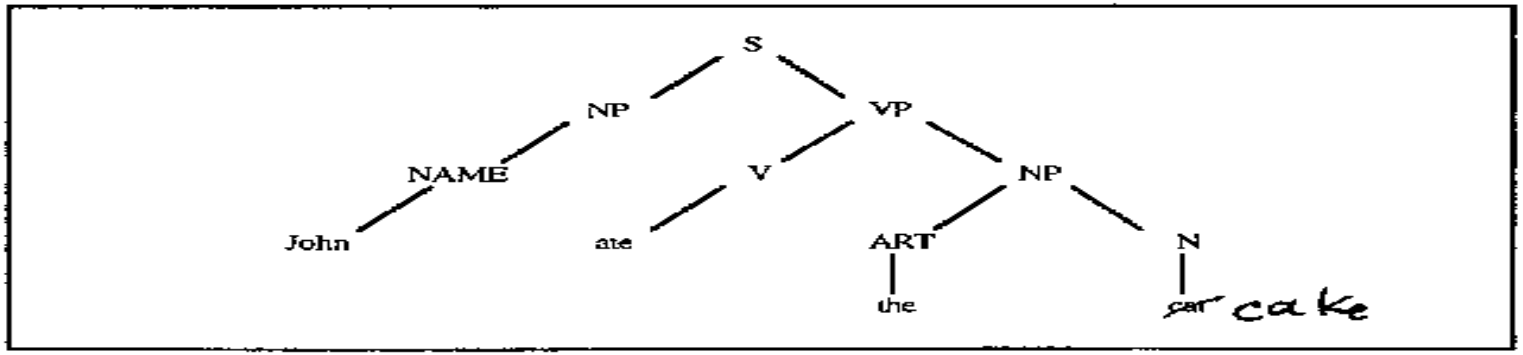


Syntactic Parsing

- Representation of a parsed sentence:

[John]NP[[ate]V[[the]ART[cat]N]NP]]

(S (NP (NAME John))
(VP (V ate)
(NP (ART the)
(N cat))))



A tree representation of *John ate the cat*

1. S → NP VP
2. VP → V NP
3. NP → NAME
4. NP → ART N
5. NAME → John
6. V → ate
7. ART → the
8. N → cat

A simple grammar

Parsing

Top-down Parsing

S	
⇒ NP VP	(rewriting S)
⇒ NAME VP	(rewriting NP)
⇒ John VP	(rewriting NAME)
⇒ John V NP	(rewriting VP)
⇒ John ate NP	(rewriting V)
⇒ John ate ART N	(rewriting NP)
⇒ John ate the N	(rewriting ART)
⇒ John ate the cat	(rewriting N)

Rules are applied from left to right.

Bottom-up Parsing

⇒ NAME ate the cat	(rewriting John)
⇒ NAME V the cat	(rewriting ate)
⇒ NAME V ART cat	(rewriting the)
⇒ NAME V ART N	(rewriting cat)
⇒ NP V ART N	(rewriting NAME)
⇒ NP V NP	(rewriting ART N)
⇒ NP VP	(rewriting V NP)
⇒ S	(rewriting NP VP)

Rules are applied from right to left.

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.

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.

Top-Down and Bottom-Up

- Top-down
  - Only searches for trees that can be answers (i.e. S's)
  - But also suggests trees that are not consistent with any of the words
- Bottom-up
  - Only forms trees consistent with the words
  - But suggests trees that make no sense globally

Control

- In both cases we left out how to keep track of the search space and how to make choices
  - Which node to try to expand next
  - Which grammar rule to use to expand a node
- One approach is called backtracking.
  - Make a choice, if it works out then fine
  - If not then back up and make a different choice

Syntactic Parsing

- Define a lexicon:

  Cried: V

  Dogs: N, V

  The: ART

- Grammar:

  S -> NP VP

  NP -> ART N

  NP -> ART ADJ N

  VP -> V

  VP -> V NP

Rewrite S into a sequence of terminals symbols. We want the result as soon as we can. A state of the parse is a pair: symbol list and a number indicating the current position in the sentence.

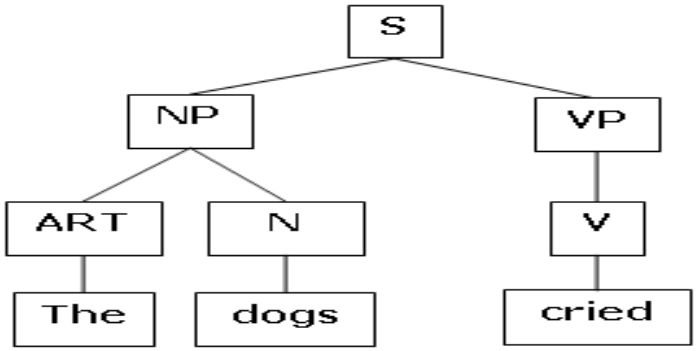
Syntactic Parsing

- 1 The 2 dogs 3 cried 4

Step	Current State	Backup States	Comment
1.	((S) 1)		initial position
2.	((NP VP) 1)		rewriting S by rule 1
3.	((ART N VP) 1)	((ART ADJ N VP) 1)	rewriting NP by rules 2 & 3
4.	((N VP) 2)	((ART ADJ N VP) 1)	matching ART with *the*
5.	((VP) 3)	((ART ADJ N VP) 1)	matching N with *dogs*
6.	((V) 3)	((V NP) 3) ((ART ADJ N VP) 1)	rewriting VP by rules 5–8
7.			the parse succeeds as V is matched to *cried*, leaving an empty grammatical symbol list with an empty sentence

Top-down depth-first parse of 1 *The* 2 *dogs* 3 *cried* 4

- Another example:
  - Parse the sentence using the same grammar with lexicon:
    1 The 2 old 3 man
    4 cried 5
    the: ART
    old: ADJ, N
    man: N, V
    cried: V

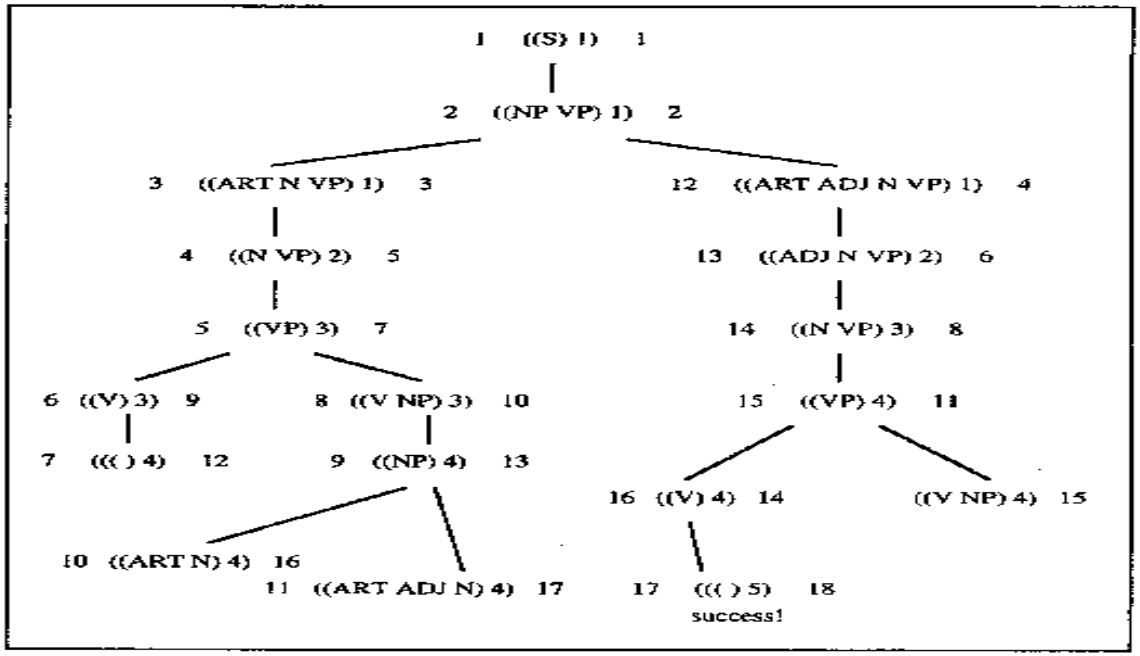


Syntactic Parsing

Step	Current State	Backup States	Comment
1.	((S) 1)		
2.	((NP VP) 1)		S rewritten to NP VP
3.	((ART N VP) 1)	((ART ADJ N VP) 1)	NP rewritten producing two new states
4.	((N VP) 2)	((ART ADJ N VP) 1)	
5.	((VP) 3)	((ART ADJ N VP) 1)	the backup state remains
6.	((V) 3)	((V NP) 3) ((ART ADJ N VP) 1)	
7.	(( ) 4)	((V NP) 3) ((ART ADJ N VP) 1)	
8.	((V NP) 3)	((ART ADJ N VP) 1)	the first backup is chosen
9.	((NP) 4)	((ART ADJ N VP) 1)	
10.	((ART N) 4)	((ART ADJ N) 4) ((ART ADJ N VP) 1)	looking for ART at 4 fails
11.	((ART ADJ N) 4)	((ART ADJ N VP) 1)	fails again
12.	((ART ADJ N VP) 1)		now exploring backup state saved in step 3
13.	((ADJ N VP) 2)		
14.	((N VP) 3)		
15.	((VP) 4)		
16.	((V) 4)	((V NP) 4)	
17.	(( ) 5)		success!

A top-down parse of 1 *The* 2 *old* 3 *man* 4 *cried* 5

Syntactic Parsing



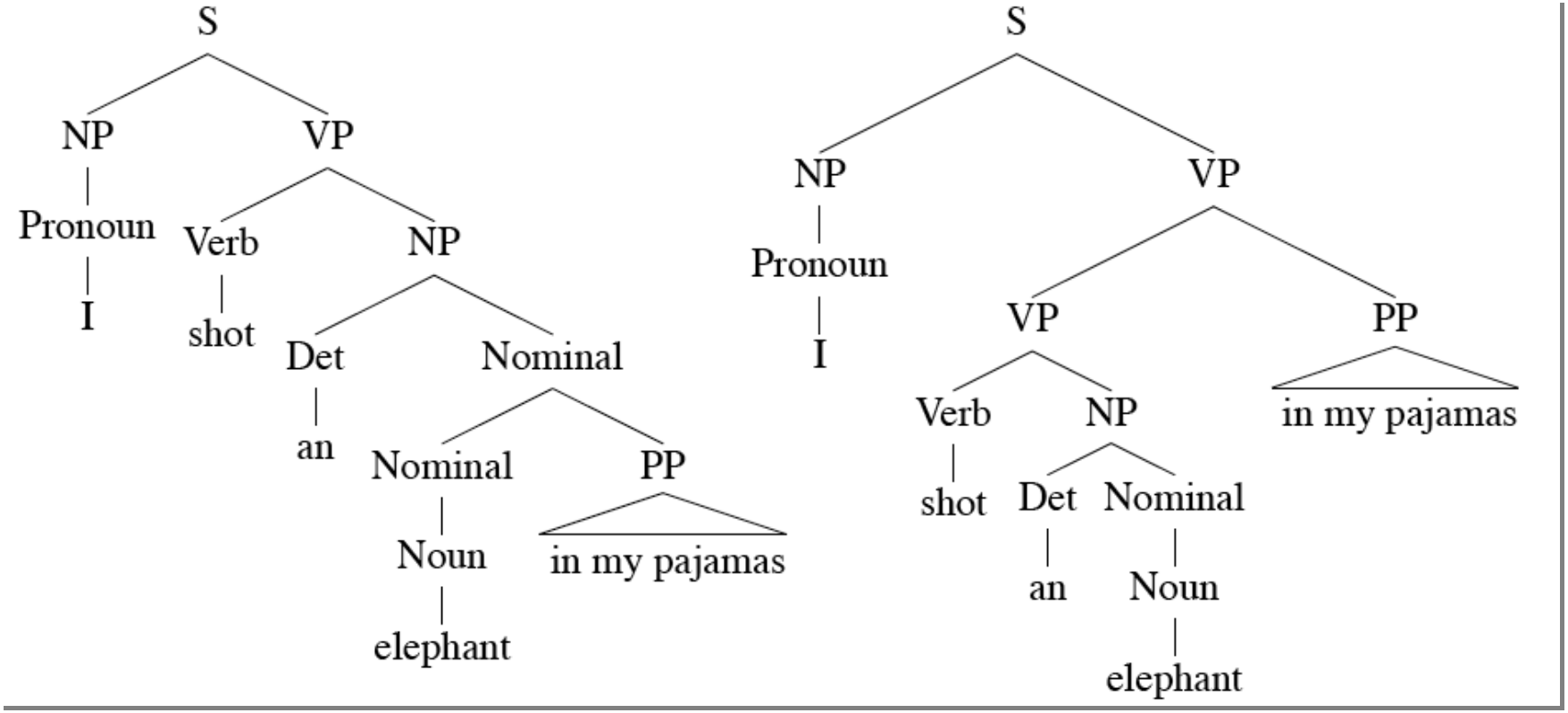
Search tree for two parse strategies (depth-first strategy on left; breadth-first on right)

- Depth-first search: the states form a stack LIFO policy.
- Breath-first search: the states form a queue FIFO policy.
  Note the large number of states for a small sentence.

Problems

- Even with the best filtering, backtracking methods are doomed because of two inter-related problems
  - Ambiguity
  - Shared subproblems

Ambiguity

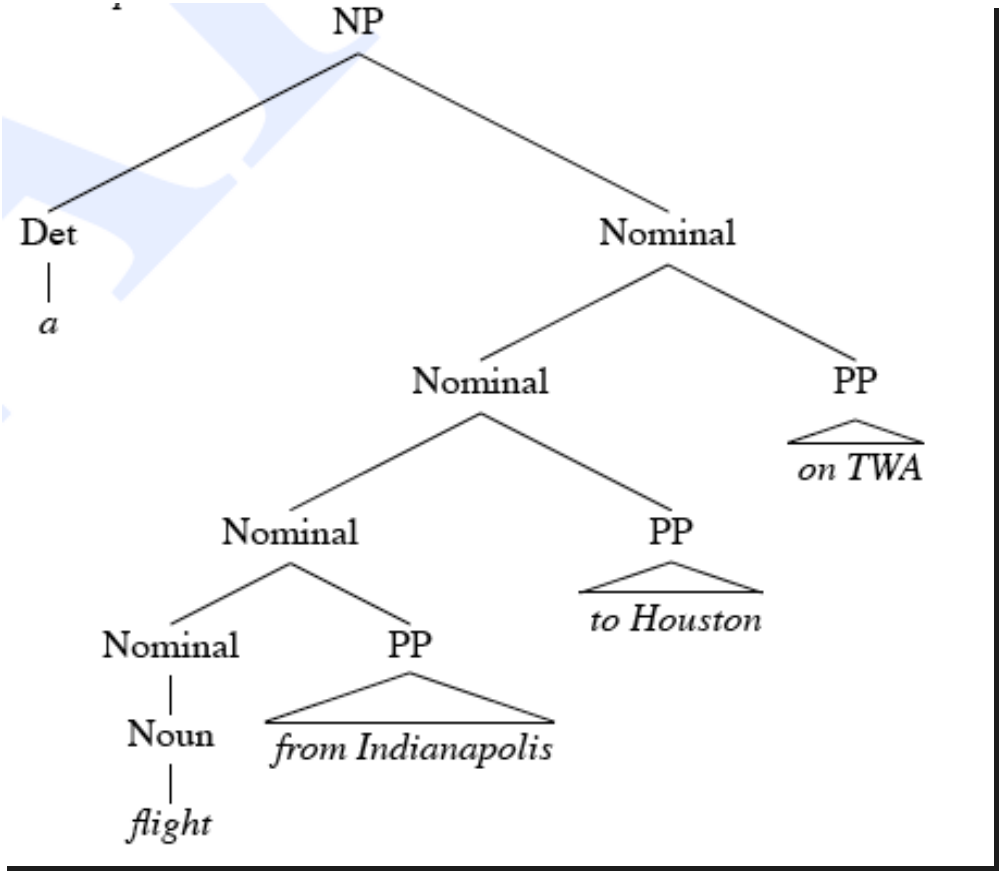


Shared Sub-Problems

- 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

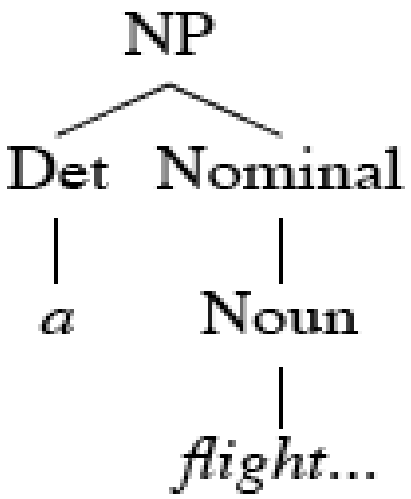
- Consider
  - A flight from Indianapolis to Houston on TWA



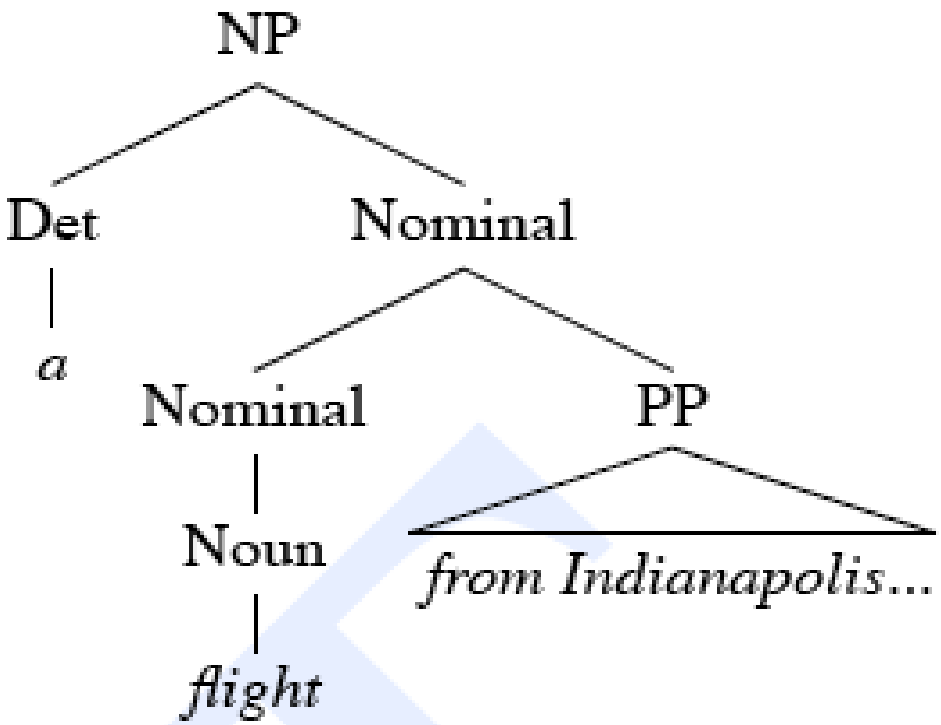
Shared Sub-Problems

- 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...

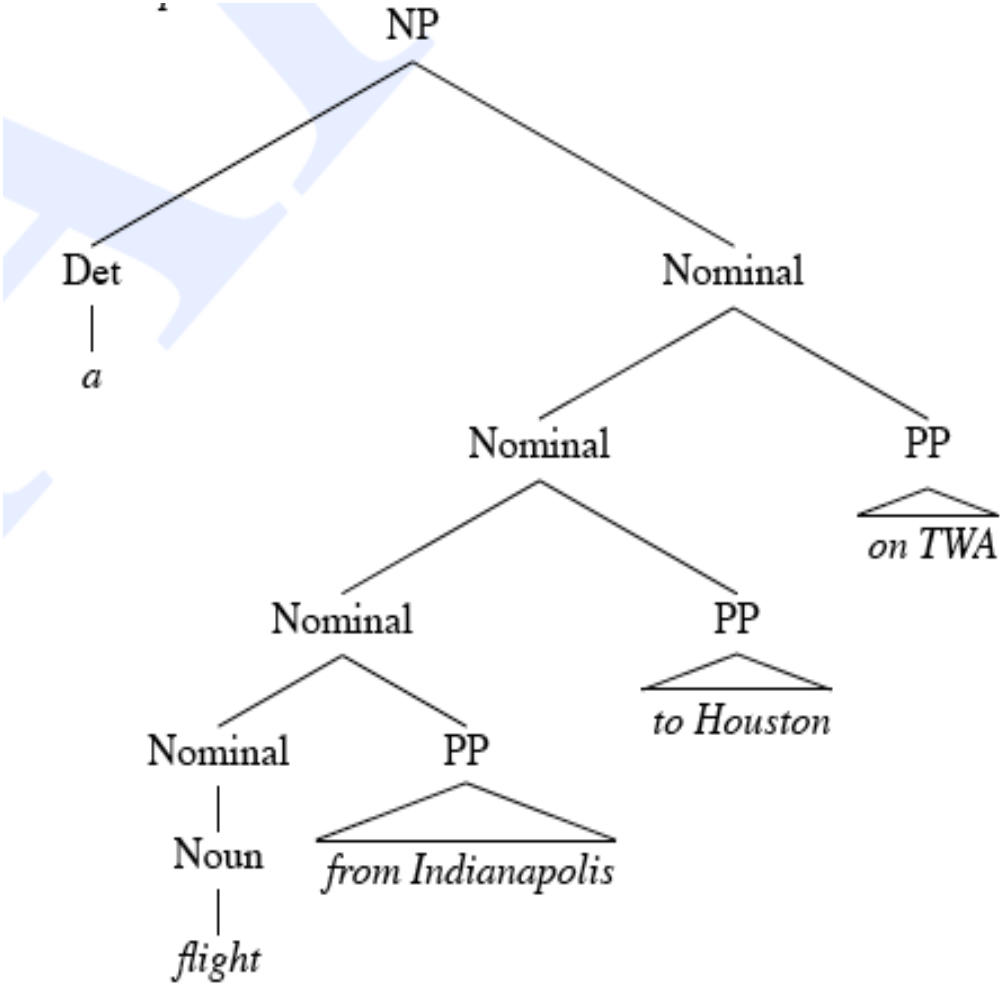
Shared Sub-Problems



Shared Sub-Problems



Shared Sub-Problems



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 $A \rightarrow BC$
  - 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 (i<k<j) where B splits from the C

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

  $A \rightarrow B\ C$

  Or

  $A \rightarrow w$

  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… $S \rightarrow A\ B\ C$ *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 L1 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

- 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 $j \leftarrow$ **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**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k],$
          $C \in table[k,j]\}$

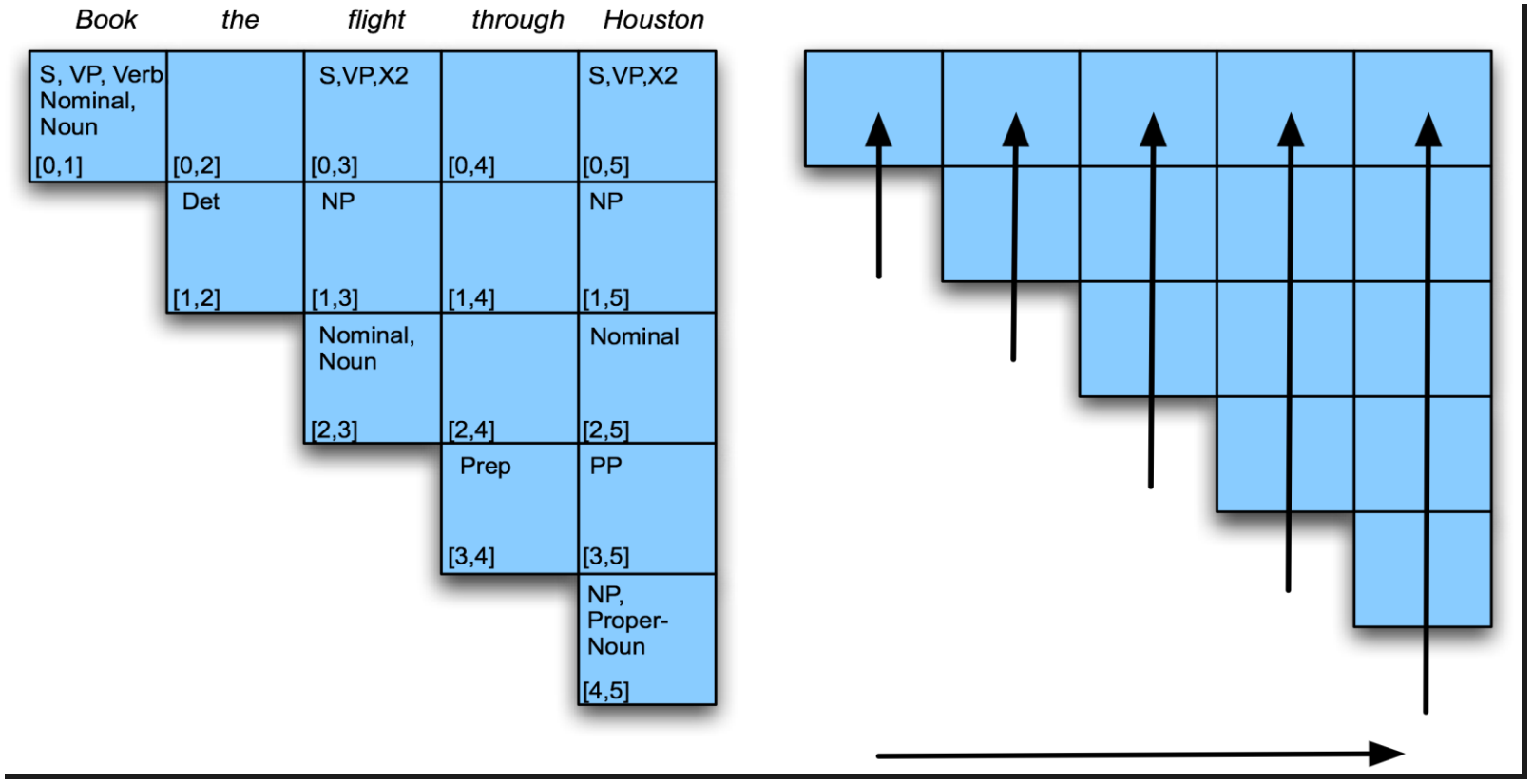
CKY Parsing

- Is that really a parser?

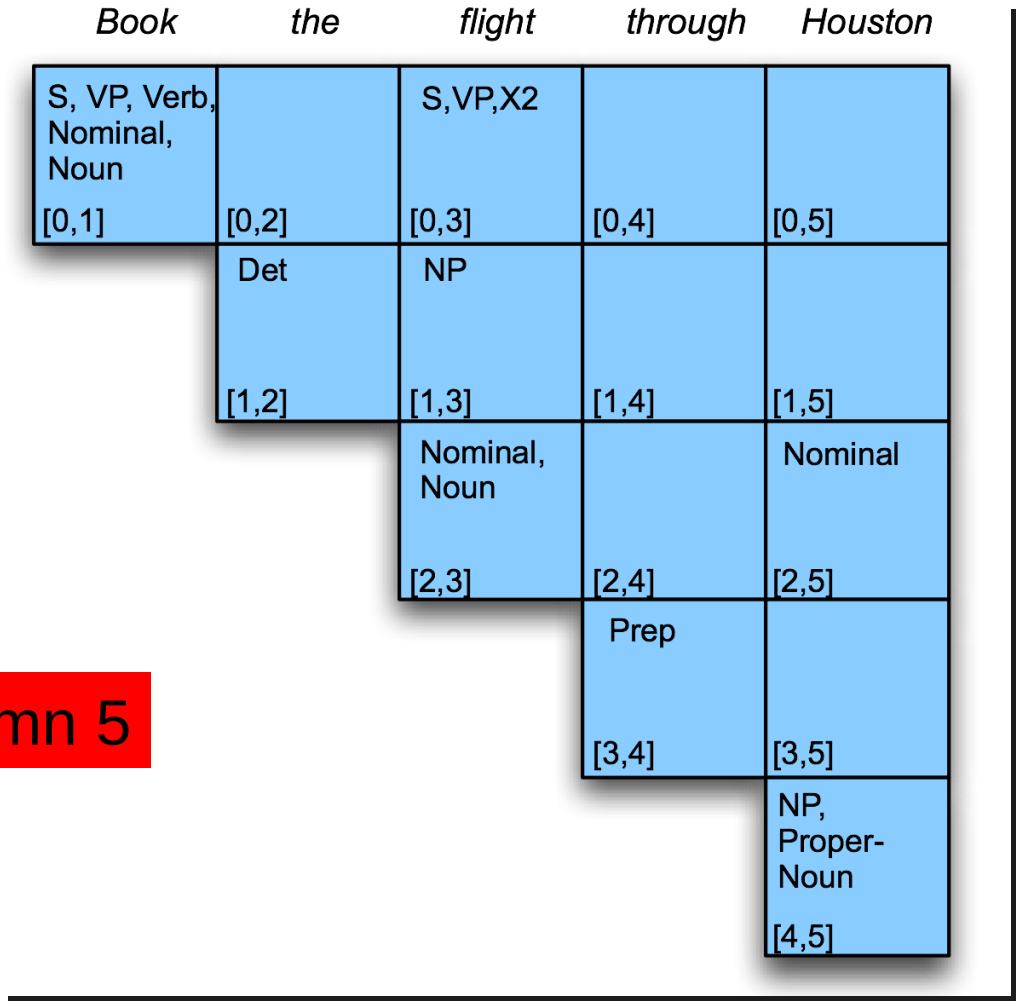
Note

- 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

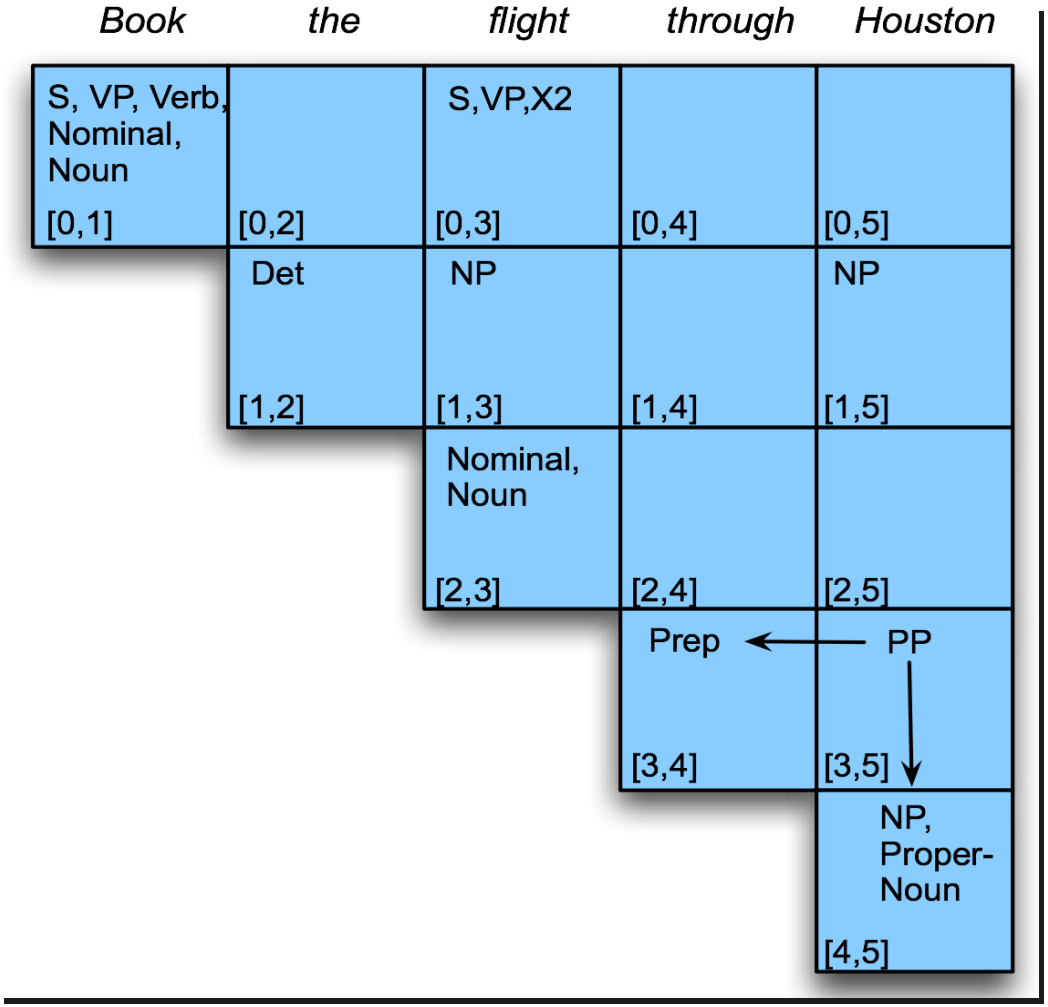


Example

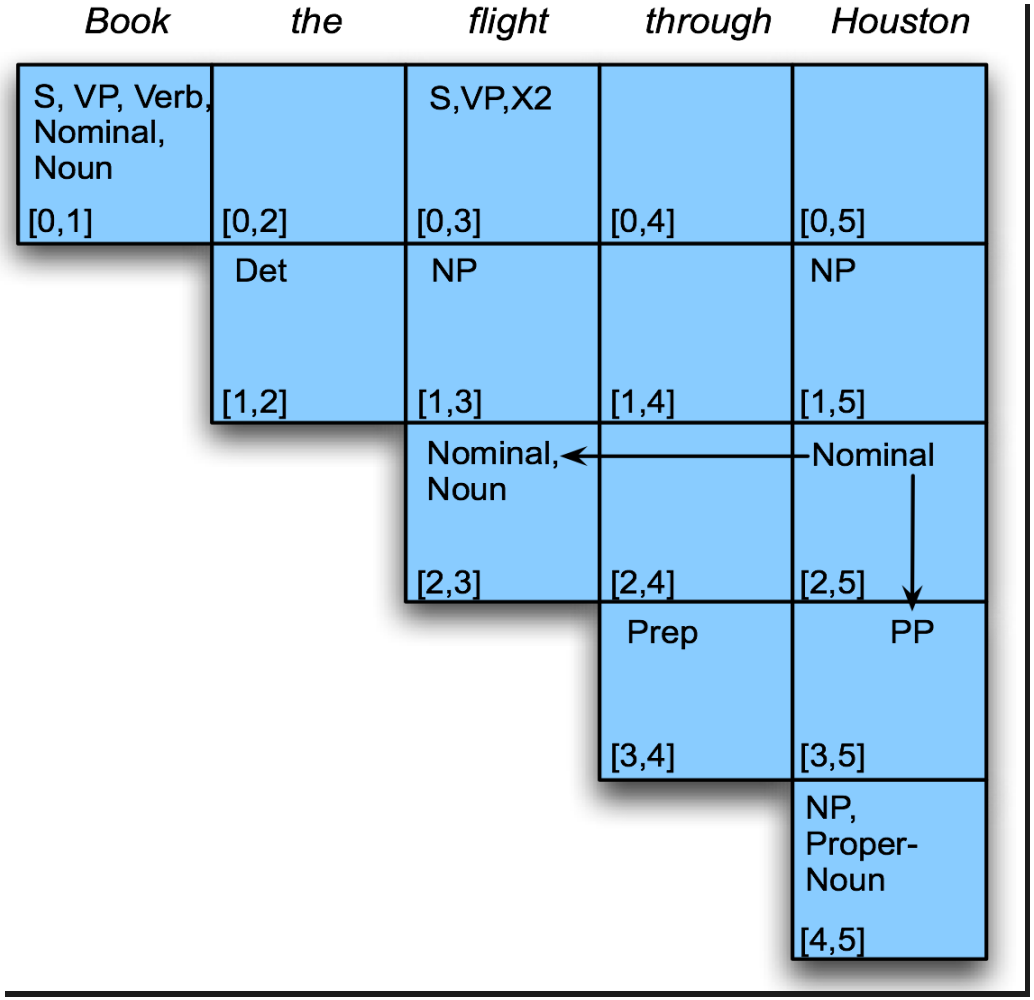


Filling column 5

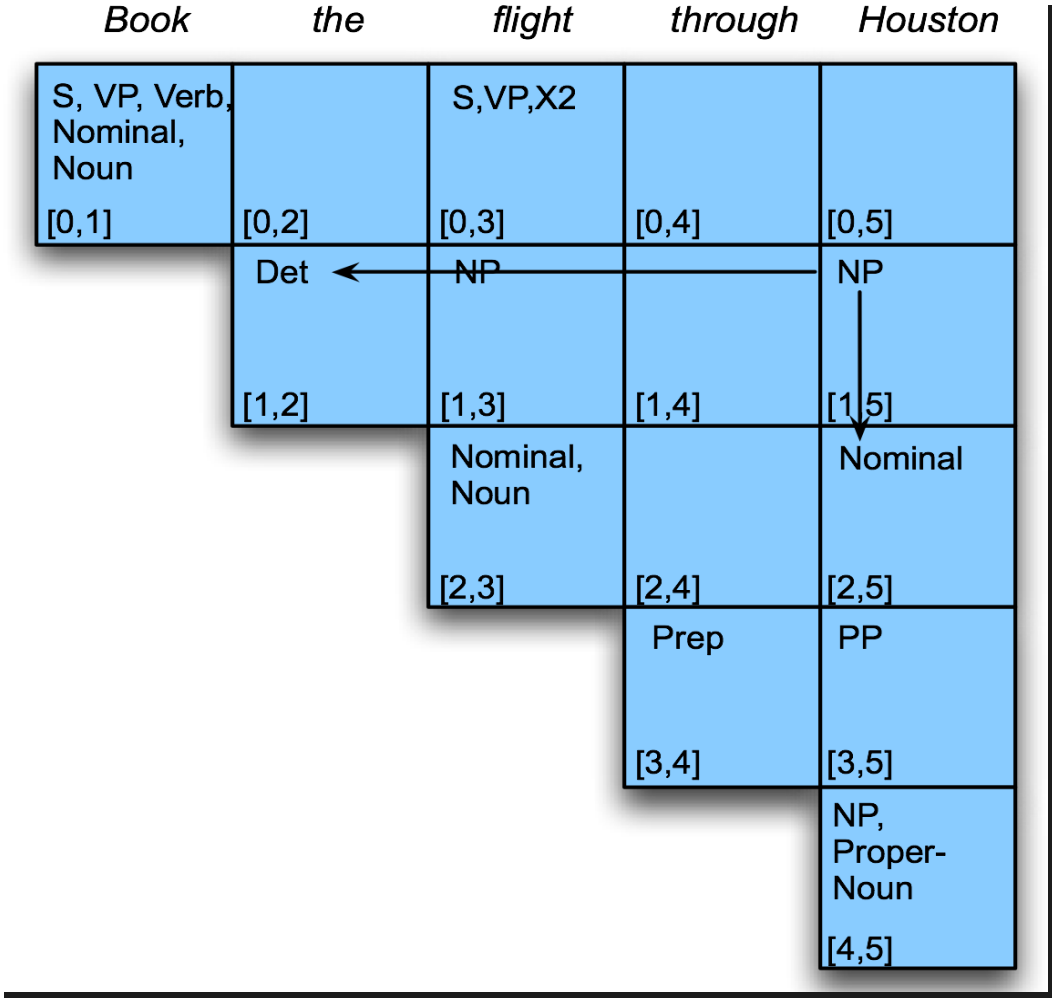
Example



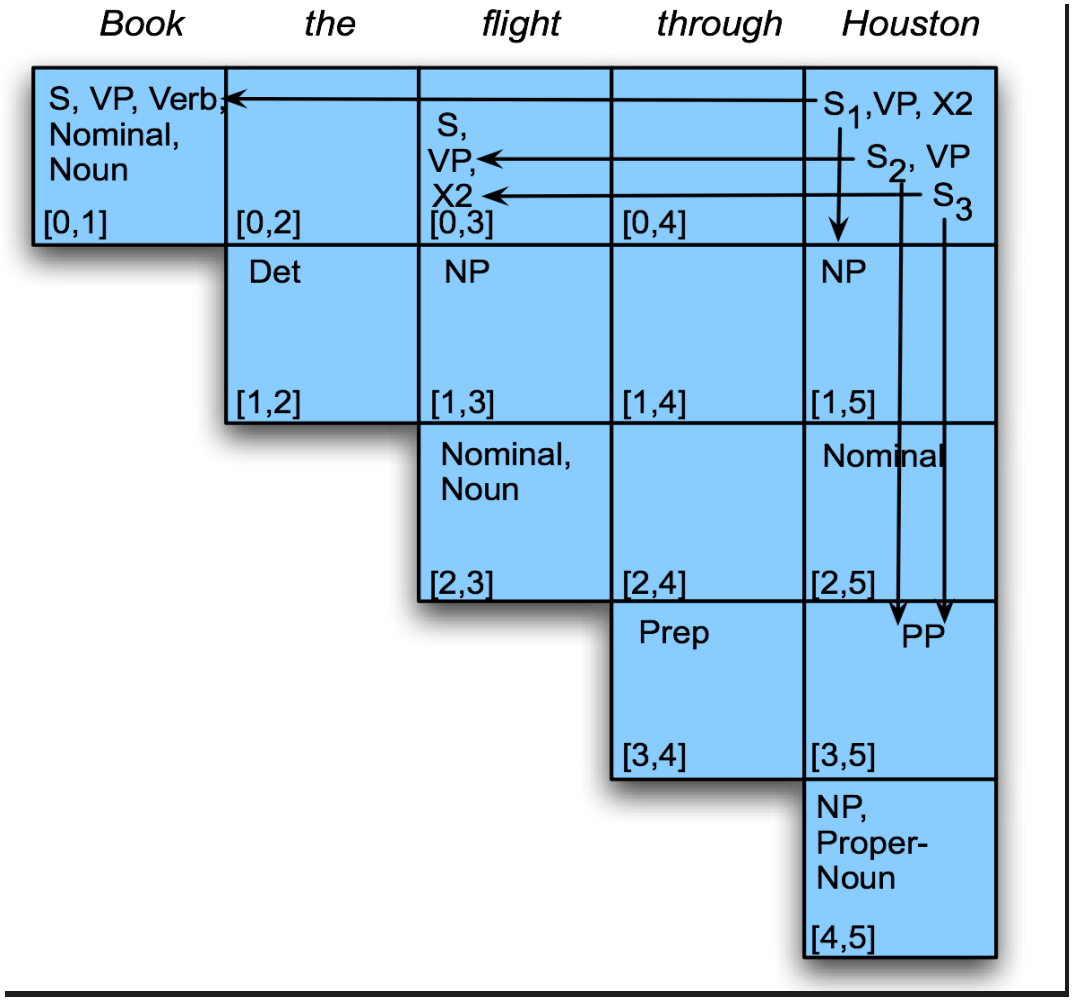
Example



Example



Example



Example 2: CYK Algorithm

The	flights	leave	on	time
[0,1]	[0,2]	[0,3]	[0,4]	[0,5]
	[1,2]	[1,3]	[1,4]	[1,5]
		[2,3]	[2,4]	[2,5]
			[3,4]	[3,5]
				[4,5]

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**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k],$
          $C \in table[k,j]\}$

LENGTH (words) = 5

Example 2: CYK Algorithm

The	flights	leave	on	time
DET [0,1]	[0,2]	[0,3]	[0,4]	[0,5]
	[1,2]	[1,3]	[1,4]	[1,5]
		[2,3]	[2,4]	[2,5]
			[3,4]	[3,5]
				[4,5]

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**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k],$
          $C \in table[k,j]\}$

LENGTH (words) = 5

j = 1
table[0,1]=DET

Example 2: CYK Algorithm

The	flights	leave	on	time
DET [0,1]	NP [0,2]	[0,3]	[0,4]	[0,5]
	Noun, Verb,VP [1,2]	[1,3]	[1,4]	[1,5]
		[2,3]	[2,4]	[2,5]
			[3,4]	[3,5]
				[4,5]

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**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k],$
          $C \in table[k,j]\}$

LENGTH (words) = 5

j = 2
table[1,2]=Noun, Verb, VP
i = 0
k = 1
table[0,2]=table[0,1] U table[1,2]
=(DET) U (Noun)
= NP

Example 2: CYK Algorithm

The	flights	leave	on	time
DET [0,1]	NP [0,2]	S [0,3]	[0,4]	[0,5]
	Noun, Verb, VP [1,2]	[1,3]	[1,4]	[1,5]
		Noun, Verb, VP [2,3]	[2,4]	[2,5]

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**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k],$
          $C \in table[k,j]\}$

j = 3
table[2,3]=Noun, Verb, VP
i = 1 to 0

i=0
k = 1 to 2
k=1
table[0,3] = table[0,1] U table[1,3]
= (DET) U (Nothing)
= NOTHING
k=2
table[0,3]= table[0,2] U table[2,3]
= (NP) U (Noun|Verb| VP)
= S

i=1
k = 2
table[1,3]= table[1,2] U table[2,3]
=(Noun|Verb) U (Noun|Verb)
= NOTHING

Example 2: CYK Algorithm

The	flights	leave	on	time
DET [0,1]	NP [0,2]	S [0,3]	[0,4]	[0,5]
	Noun, Verb [1,2]	[1,3]	[1,4]	[1,5]
		Noun, Verb, VP [2,3]	[2,4]	[2,5]
			Prep [3,4]	[3,5]
				[4,5]

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**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k],$
          $C \in table[k,j]\}$

j = 4
table[3,4]=Prep
i = 2 to 0

i=2 k = 3
table[2,4]= table[2,3] U table[3,4]
= NOTHING

i=1 k = 2 to 3 k=2
table[1,4]= table[1,2] U table[2,4]
= NOTHING

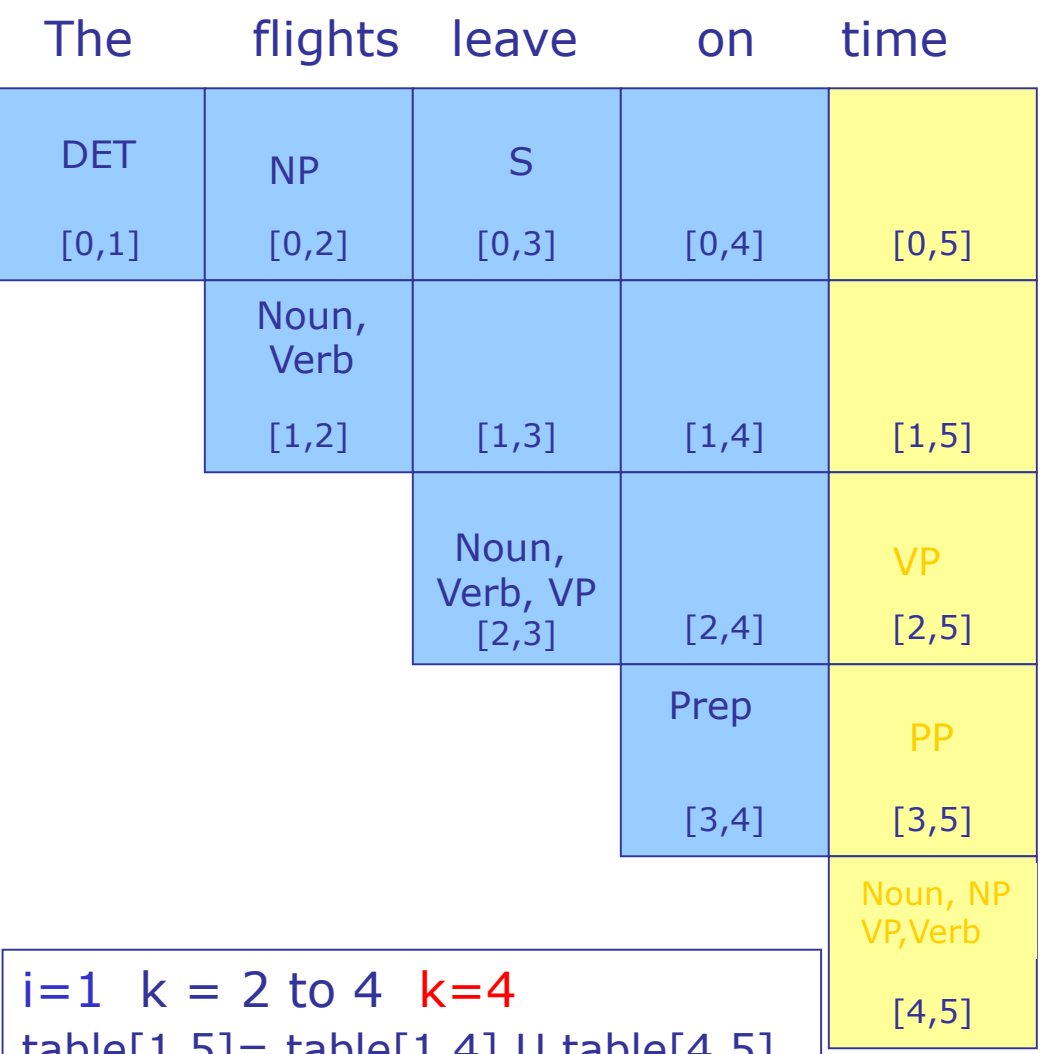
i=1 k = 2 to 3 k=3
table[1,4]= table[1,3] U table[3,4]
= NOTHING

i=0 k = 1 to 3 k=1
table[0,4]= table[0,1] U table[1,4]
= NOTHING

i=0 k = 1 to 3 k=2
table[0,4]= table[0,2] U table[2,4]
= NOTHING

i=0 k = 1 to 3 k=3
table[0,4]= table[0,3] U table[3,4]
= NOTHING

Example 2: CYK Algorithm



j = 5
table[3,4]=Noun|NP|Verb|VP
i = 3 to 0

i=3 k = 4
table[3,5]= table[3,4] U table[4,5]
= PP

i=2 k = 3 to 4 k=3
table[2,5]= table[2,3] U table[3,5]
= VP

i=2 k = 3 to 4 k=4
table[2,5]= table[2,4] U table[4,5]
= NOTHING

i=1 k = 2 to 4 k=2
table[1,5]= table[1,2] U table[2,5]
= NOTHING

i=1 k = 2 to 4 k=3
table[1,5]= table[1,3] U table[3,5]
= NOTHING

i=1 k = 2 to 4 k=4
table[1,5]= table[1,4] U table[4,5]
= NOTHING

Example 2: CYK Algorithm

The	flights	leave	on	time
DET [0,1]	NP [0,2]	S [0,3]	[0,4]	**S** [0,5]
	Noun, Verb [1,2]	[1,3]	[1,4]	[1,5]
		Noun, Verb, VP [2,3]	[2,4]	VP [2,5]
			Prep [3,4]	PP [3,5]
				Noun, NP VP,Verb [4,5]

j = 5
table[3,4]=Noun|NP|Verb|VP
i = 3 to 0

i=0 k = 1 to 4
k=1
table[0,5]= table[0,1] U table[1,5]
= NOTHING

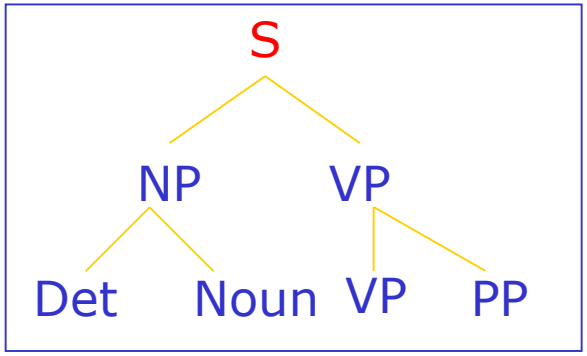
i=0 k = 1 to 4
k=2
table[0,5]= table[0,2] U table[2,5]
= S

i=0 k = 1 to 4
k=3
table[0,5]= table[0,3] U table[3,5]
= NOTHING

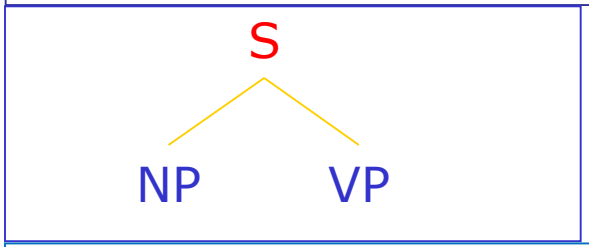
i=0 k = 1 to 4
k=1
table[0,5]= table[0,4] U table[4,5]
= NOTHING

Example 2: CYK Algorithm

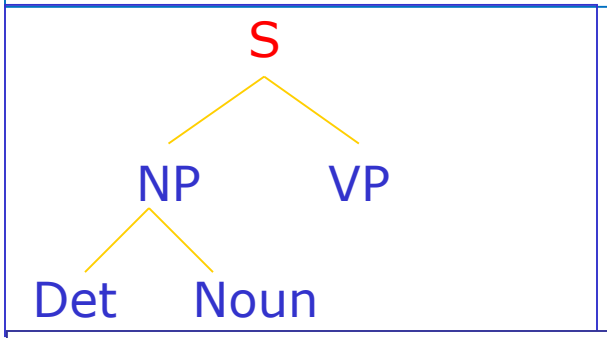
The	flights	leave	on	time
DET [0,1]	NP [0,2]	S [0,3]	[0,4]	**S** [0,5]
	Noun, Verb [1,2]	[1,3]	[1,4]	[1,5]
		Noun, Verb, VP [2,3]	[2,4]	VP [2,5]
			Prep [3,4]	PP [3,5]
				Noun, NP VP,Verb [4,5]



i=0 k=2
table[0,5]= **S**
table[0,2] U table[2,5]
NP U VP

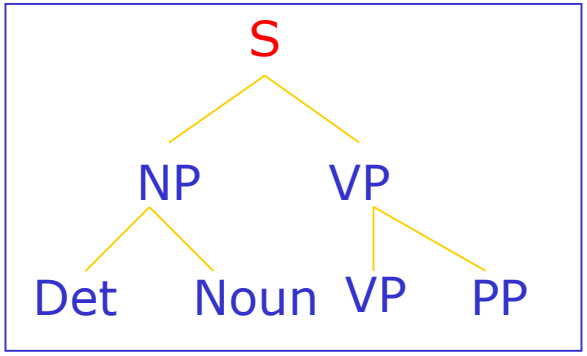


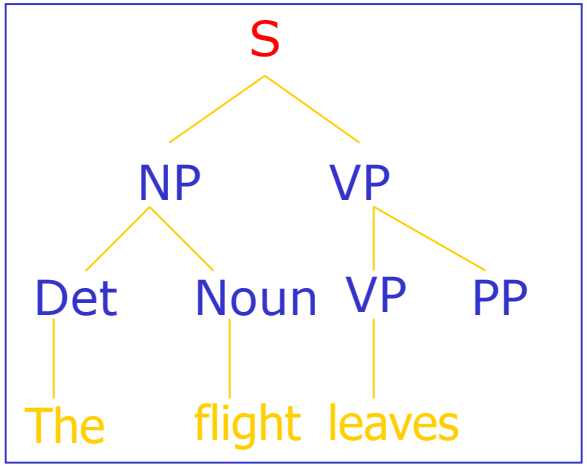
table[0,2]= NP
table[0,1] U table[1,2]
DET U Noun

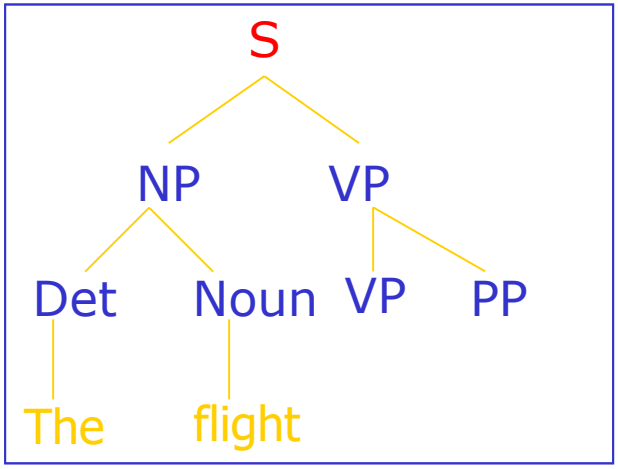


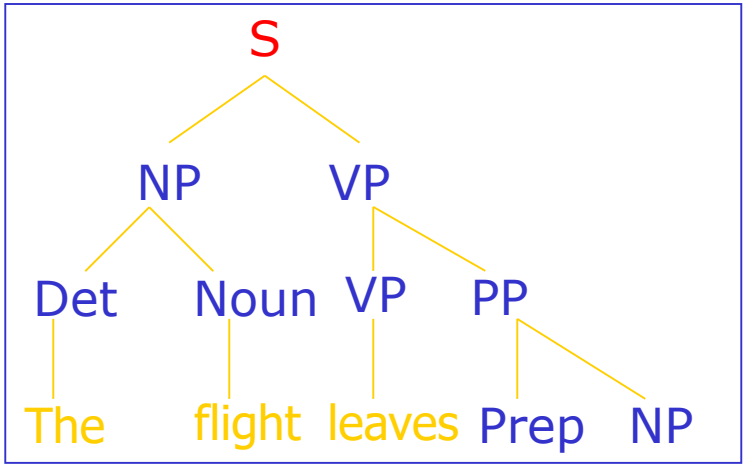
table[2,5]= VP
table[2,3] U table[3,5]
VP U PP

Example 2: What is the parse ???

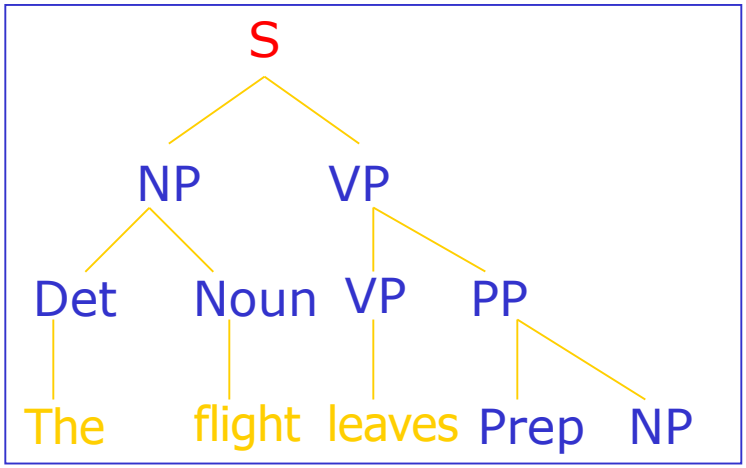


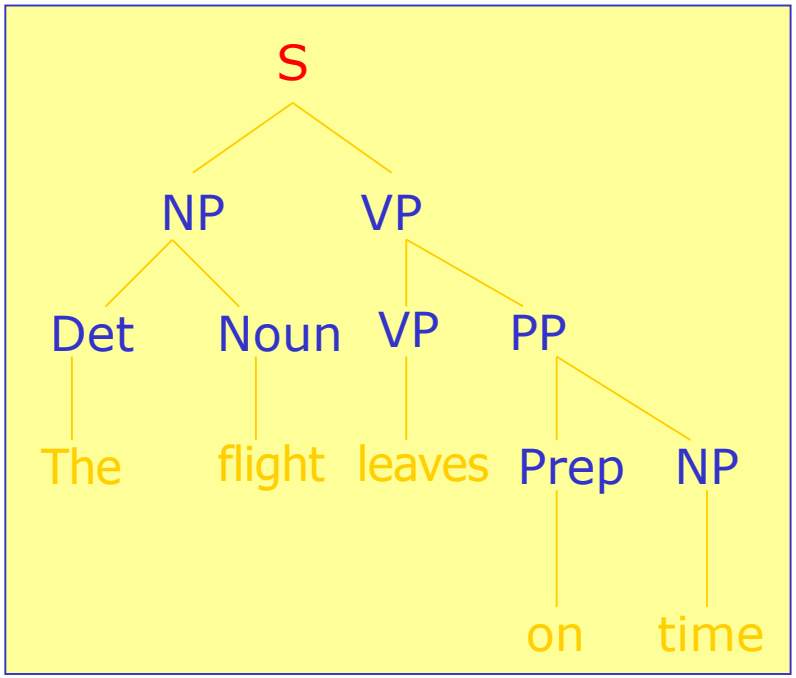






Final Parse


S
NP
VP
Det
Noun
VP
PP
The
flight
leaves
Prep
NP


S
NP
VP
Det
Noun
VP
PP
The
flight
leaves
Prep
NP
on
time

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

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

- 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

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 and is complete. That is,
  - $S \rightarrow \alpha \square [0,N]$
- If that's the case you're done.

Earley

- So sweep through the table from 0 to N…
  - New predicted states are created by starting top-down from S
  - 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. Generate new predictions
     3. Go to step 2
  3. When you're out of words, look at the chart to see if you have a winner

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

Example

- Book that flight
- We should find… an S from 0 to 3 that is a completed state…

Chart[0]

S0	$\gamma \rightarrow \bullet S$	[0,0]	Dummy start state
S1	$S \rightarrow \bullet NP\ VP$	[0,0]	Predictor
S2	$S \rightarrow \bullet Aux\ NP\ VP$	[0,0]	Predictor
S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
S5	$NP \rightarrow \bullet Proper\text{-}Noun$	[0,0]	Predictor
S6	$NP \rightarrow \bullet Det\ Nominal$	[0,0]	Predictor
S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
S8	$VP \rightarrow \bullet Verb\ NP$	[0,0]	Predictor
S9	$VP \rightarrow \bullet Verb\ NP\ PP$	[0,0]	Predictor
S10	$VP \rightarrow \bullet Verb\ PP$	[0,0]	Predictor
S11	$VP \rightarrow \bullet VP\ PP$	[0,0]	Predictor

Note that given a grammar, these entries are the same for all inputs; they can be pre-loaded.

Chart[1]

S12	*Verb* → *book* •	[0,1]	Scanner
S13	*VP* → *Verb* •	[0,1]	Completer
S14	*VP* → *Verb* • *NP*	[0,1]	Completer
S15	*VP* → *Verb* • *NP PP*	[0,1]	Completer
S16	*VP* → *Verb* • *PP*	[0,1]	Completer
S17	*S* → *VP* •	[0,1]	Completer
S18	*VP* → *VP* • *PP*	[0,1]	Completer
S19	*NP* → • *Pronoun*	[1,1]	Predictor
S20	*NP* → • *Proper-Noun*	[1,1]	Predictor
S21	*NP* → • *Det Nominal*	[1,1]	Predictor
S22	*PP* → • *Prep NP*	[1,1]	Predictor

Charts[2] and [3]

S23	$Det \rightarrow that \bullet$	[1,2]	Scanner
S24	$NP \rightarrow Det \bullet Nominal$	[1,2]	Completer
S25	$Nominal \rightarrow \bullet Noun$	[2,2]	Predictor
S26	$Nominal \rightarrow \bullet Nominal\ Noun$	[2,2]	Predictor
S27	$Nominal \rightarrow \bullet Nominal\ PP$	[2,2]	Predictor
S28	$Noun \rightarrow flight \bullet$	[2,3]	Scanner
S29	$Nominal \rightarrow Noun \bullet$	[2,3]	Completer
S30	$NP \rightarrow Det\ Nominal \bullet$	[1,3]	Completer
S31	$Nominal \rightarrow Nominal \bullet Noun$	[2,3]	Completer
S32	$Nominal \rightarrow Nominal \bullet PP$	[2,3]	Completer
S33	$VP \rightarrow Verb\ NP \bullet$	[0,3]	Completer
S34	$VP \rightarrow Verb\ NP \bullet PP$	[0,3]	Completer
S35	$PP \rightarrow \bullet Prep\ NP$	[3,3]	Predictor
S36	$S \rightarrow VP \bullet$	[0,3]	Completer
S37	$VP \rightarrow VP \bullet PP$	[0,3]	Completer

Summary

Chart[1]	S12	*Verb* → *book* •	[0,1]	Scanner
Chart[2]	S23	*Det* → *that* •	[1,2]	Scanner
Chart[3]	S28	*Noun* → *flight* •	[2,3]	Scanner
	S29	*Nominal* → *Noun* •	[2,3]	(S28)
	S30	*NP* → *Det Nominal* •	[1,3]	(S23, S29)
	S33	*VP* → *Verb NP* •	[0,3]	(S12, S30)
	S36	*S* → *VP* •	[0,3]	(S33)

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.

Details

- As with CKY that isn't a parser until we add the backpointers so that each state knows where it came from.

Back to Ambiguity

- Did we solve it?

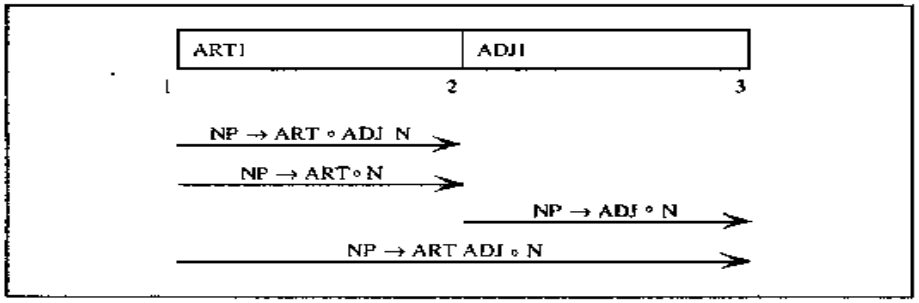
Ambiguity

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

Example

- A bottom up chart parser:
  - The idea is to match a sequence of symbols to the right hand side of each rule to determine if a rule is applicable.
  - To reduce the search space use a data structure called a chart that keeps track of successful rules. The process stops when the entire sentence is covered. Efficiency results from not repeating the construction of sentence blocks (or constituents).
- Grammar:

1. S → NP VP
2. NP → ART ADJ N
3. NP → ART N
4. NP → ADJ N
5. VP → AUX VP
6. VP → V NP



The chart after seeing an ADJ in position 2

A Bottom-Up Chart Parser

- The main difference between top-down and bottom-up parser is the way the grammar rules are used
- The basic operation in bottom-up parsing is to take a sequence of symbols and match it to the right-hand side of the rules
  - rewrite a word by its possible lexical categories
  - replace a sequence of symbols that matches the right-hand side of the grammar rule by its left-hand side symbol
  - use a chart structure to keep track of the partial results, so that the work need not be reduplicated

A Bottom-Up Chart Parser (The Algorithm)

To add a constituent C from position p_1 to p_2:

1. Insert C into the chart from position p_1 to p_2.
2. For any active arc of the form $X \rightarrow X_1 \ldots \circ C \ldots X_n$ from position p_0 to p_1, add a new active arc $X \rightarrow X_1 \ldots C \circ \ldots X_n$ from position p_0 to p_2.
3. For any active arc of the form $X \rightarrow X_1 \ldots X_n \circ C$ from position p_0 to p_1, then add a new constituent of type X from p_0 to p_2 to the agenda.

Figure 3.10 The arc extension algorithm

Do until there is no input left:

1. If the agenda is empty, look up the interpretations for the next word in the input and add them to the agenda.
2. Select a constituent from the agenda (let's call it constituent C from position p_1 to p_2).
3. For each rule in the grammar of form $X \rightarrow C\ X_1 \ldots X_n$, add an active arc of form $X \rightarrow \circ\ C\ X_1 \ldots X_n$ from position p_1 to p_2.
4. Add C to the chart using the arc extension algorithm above.

Figure 3.11 A bottom-up chart parsing algorithm

A Bottom-Up Chart Parser (An Example) 1/4

- Let's consider the sentence to be parsed:
  - $_1$ *The* $_2$ *large* $_3$ *can* $_4$ *can* $_5$ *hold* $_6$ *the* $_7$ *water* $_8$
- Lexicon:
  - the: ART
  - large: ADJ
  - can: N, AUX, V
  - hold: N, V
  - water: N,V
- Grammar:

1. S → NP VP
2. NP → ART ADJ N
3. NP → ART N
4. NP → ADJ N
5. VP → AUX VP
6. VP → V NP

A Bottom-Up Chart Parser (An Example) 2/4

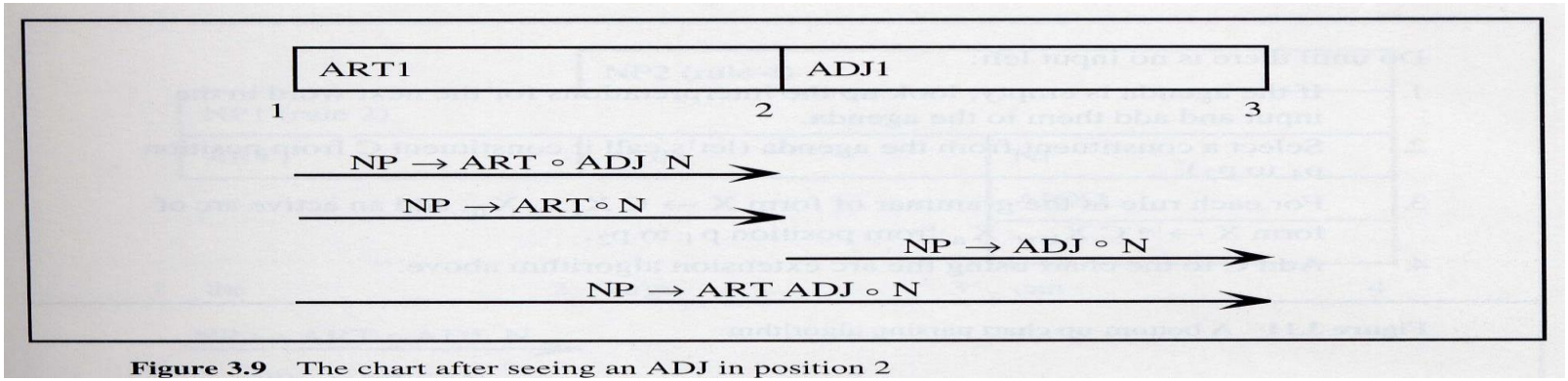


Figure 3.9 The chart after seeing an ADJ in position 2

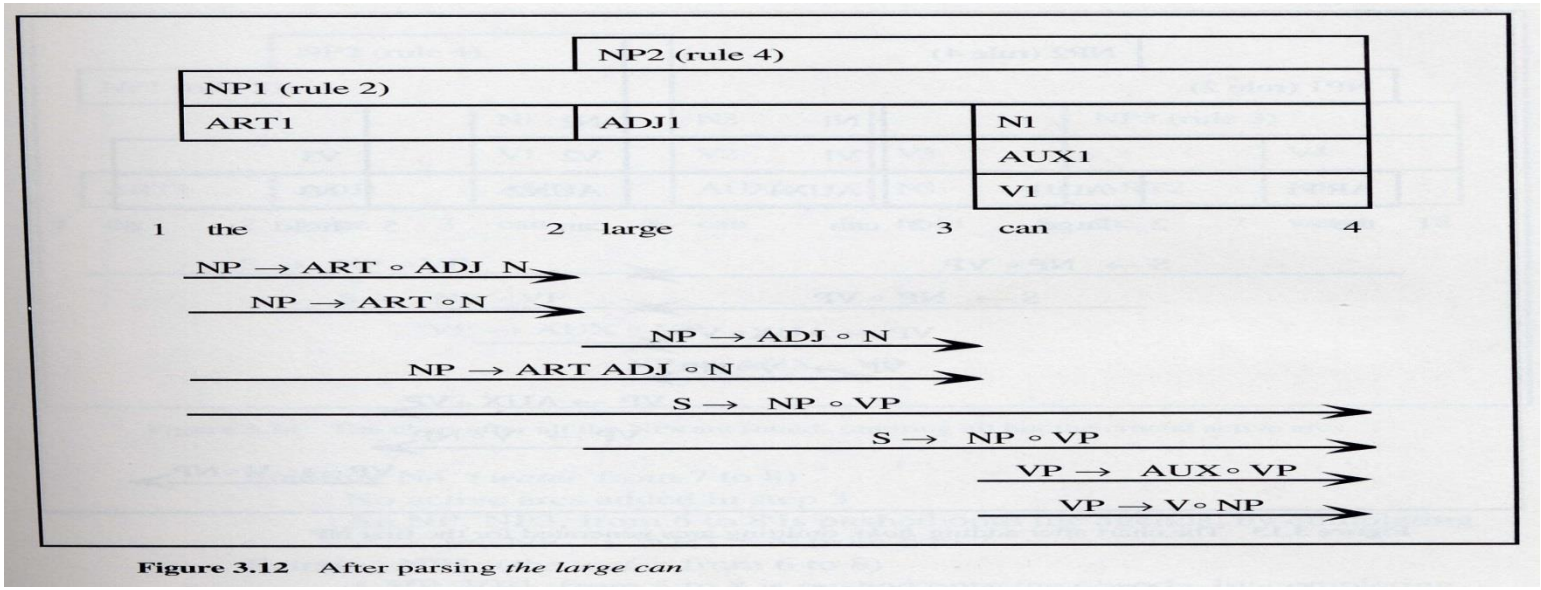


Figure 3.12 After parsing *the large can*

A Bottom-Up Chart Parser (An Example) 3/4

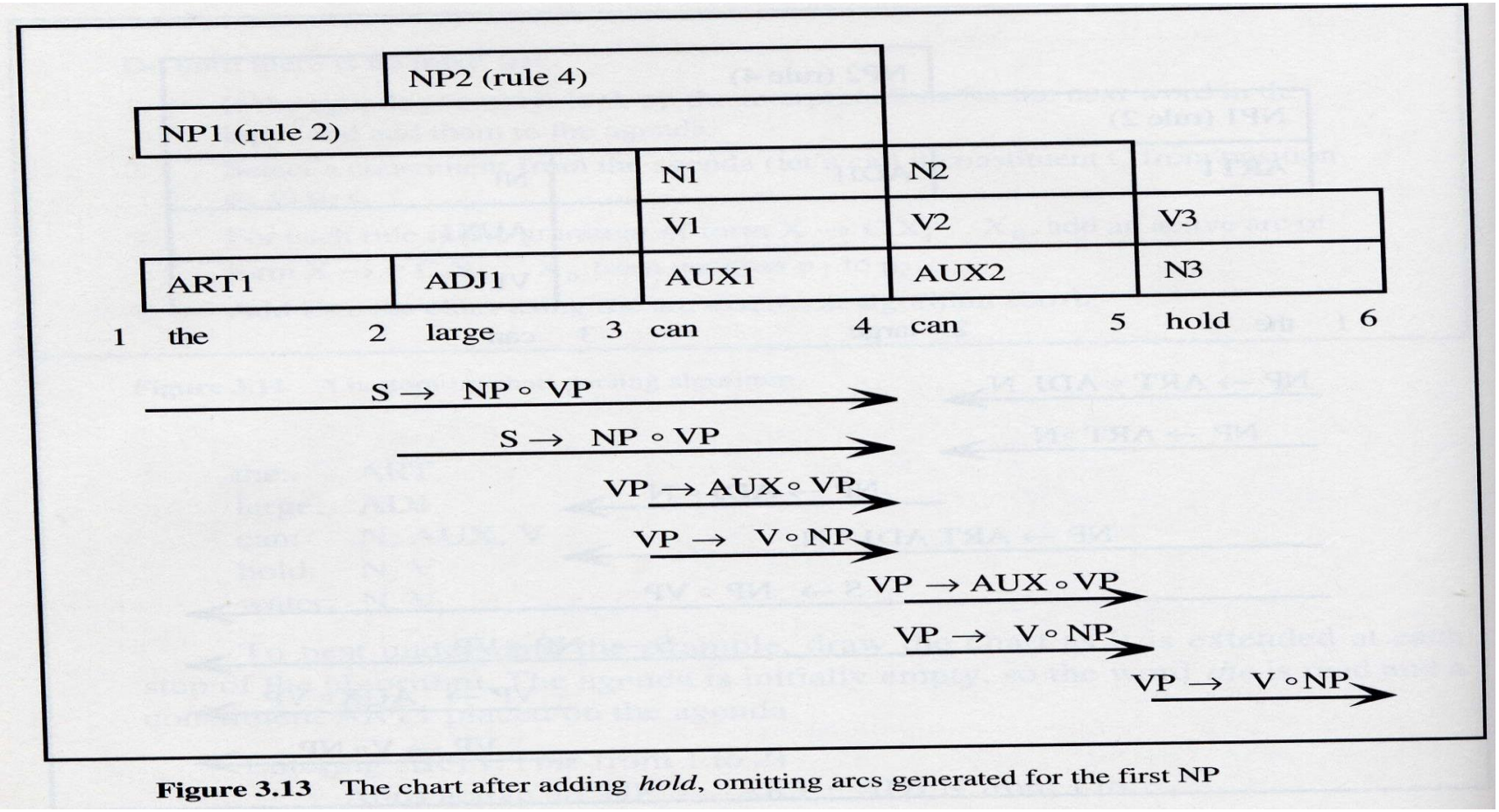


Figure 3.13 The chart after adding *hold*, omitting arcs generated for the first NP

A Bottom-Up Chart Parser (An Example) 4/4

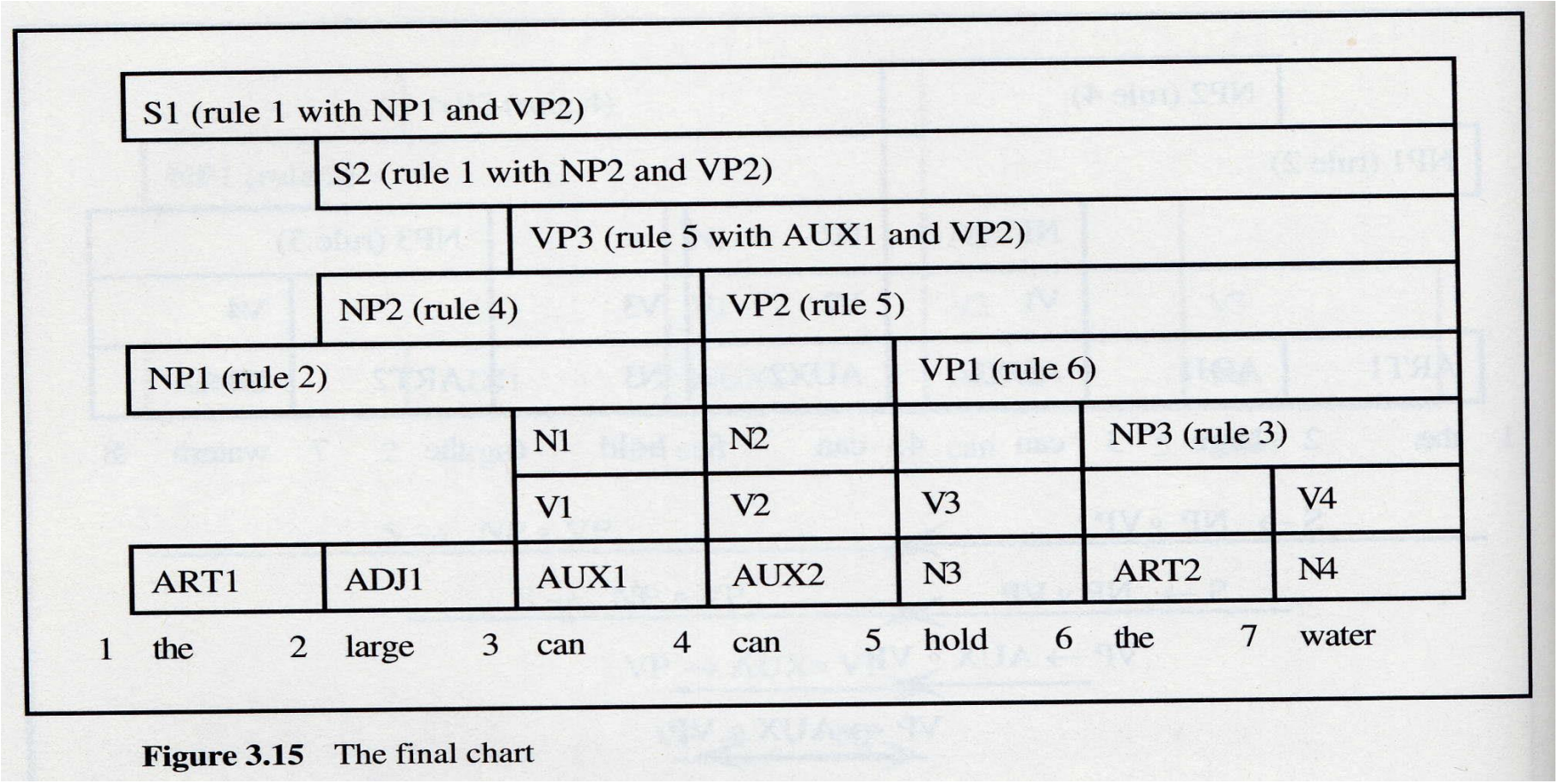


Figure 3.15 The final chart

Efficiency Considerations

- A pure top-down or bottom-up search strategy could require up to C^n operations to parse a sentence of length n
- A chart-based parser, in the worst case would build every possible constituent between every possible pair of positions => a worst case complexity of $K*n^3$
- A chart parser involves more work in each step: K >> C
- Let's say that C is 10, K is 1000 and the sentence to be parsed has 12 words:
  - Brute force search might take 10^{12} operations (1,000,000,000,000)
  - Chart parser would take $1000*12^3$ (1,728,000)
  - Under these assumptions, the chart parser would be up to 500,000 times faster than the brute force search on some examples!!!