Syntactic Parsing

Syntax

Syntax

- **Syntax** (of natural languages) describe how words are strung together to form components of sentences, and how those components are strung together to form sentences.

- In the core of the description of the syntax of a natural language, we use context-free grammars (CFGs).

- Groups of words may behave as a single unit or phrases, called as **constituent**.
  - noun phrase,
  - verb phrase

- CFGs will allow us to model these constituency facts.

Constituency

- **How do words group together?**
- Some noun phrases (noun groups) in English:
  - three parties from Brooklyn they three books
- All noun phrases can appear in similar syntactic environments:
  - three parties from Brooklyn arrive they arrive
- Although whole noun phrase can appear before a verb, its parts may not appear before verb.
  - * from arrive
- A noun phrase can be placed in certain places in a sentence.
- A prepositional phrase can be placed in different places in the sentences.
  - On September seventh, I'd like to fly from Atlanta to Denver
  - I'd like to fly on September seventh from Atlanta to Denver
  - I'd like to fly from Atlanta to Denver on September seventh

Context Free Grammars

- CFGs capture constituency and ordering in natural language sentences.
- But we will need extra information to model:
  - grammatical relations such as agreement
  - subcategorization of verbs
  - dependency relations between words and phrases
- So, a CFG will be in the core of the description of the syntax of a natural language.
- Context-Free Grammars are also called as **Phrase-Structure Grammars**.

Why not Other Formalisms

- Why do we use CFG to describe the syntax of a natural language.
  - Regular Grammars -- too weak
  - Context Sensitive Grammars -- too strong.
  - Turing Machines -- way too strong.

- Too weak means that they cannot capture/describe the syntactic structures which exist in natural languages.

- Too strong means that we do not need that much power to capture/describe the syntactic structures which exist in natural languages.

- For weaker methods, we have much efficient computational processes.

Definition of CFG

- A CFG consists of:
  - **Sets of terminals** (either lexical items or parts of speech)
  - **Sets of non-terminals** (the constituents of the language)
  - **Sets of rules of the form** $\mathbf{A \rightarrow \alpha}$ where α is a string of zero or more terminals and non-terminals.
  - One of non-terminals is designated as a **start symbol**.

An Example of CFG

S → NP VP

NP → Pronoun | NOM | Det NOM

NOM → Noun | Noun NOM

VP → Verb NP

-- Lexicon -- (parts of speech)

Prounoun → I | they

Noun → flight | morning | evening

Verb → prefer

Det → a | the | that

Derivation

- A **derivation** is a sequence of rule applications.
- In each rule application, a non-terminal in a string is re-written as α if there is a rule in the form A → α.

  $$\beta A \gamma \Rightarrow \beta \alpha \gamma$$

- We say that α_1 **derives** α_m ($\alpha_1 \Rightarrow^* \alpha_m$) if:

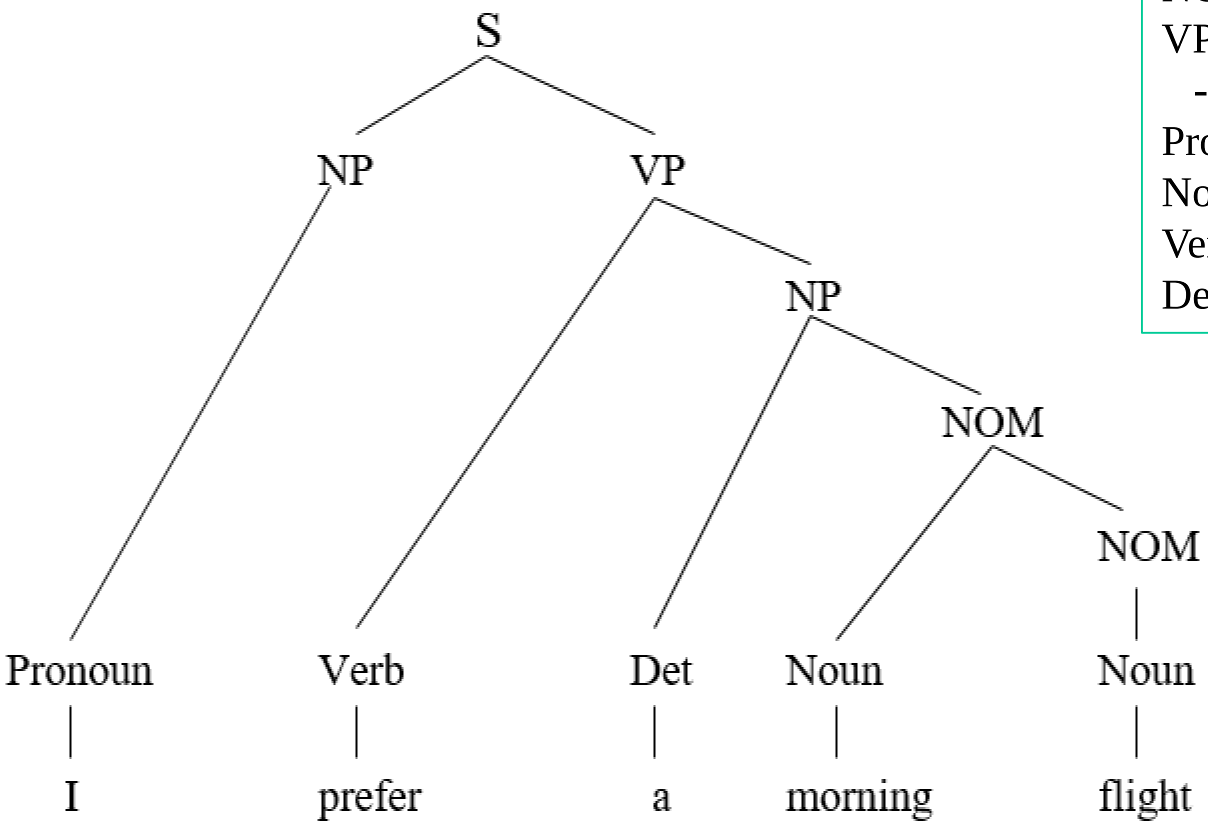
  $$\alpha_1 \Rightarrow \dots \Rightarrow \alpha_m$$

- The language generated by a CFG G is:

  L_G = { w | w is a string of terminals and S derives w }

- A derivation can be represented by a **parse tree**.
- Mapping from a string of terminals to its parse tree is called as **parsing**.

Parse Tree



S → NP VP
NP → Pronoun | NOM | Det NOM
NOM → Noun | Noun NOM
VP → Verb NP
-- Lexicon -- (parts of speech)
Prounoun → I | they
Noun → flight | morning | evening
Verb → prefer
Det → a | the | that

Developing Grammars

- We have to do a lot to develop grammars for natural languages.
    - We will look some trivial parts of grammars.

- Here we look at some constituents (syntactic substructures) in natural languages.

- The key constituents are: (We will investigate)
    - Sentences
    - Noun Phrases
    - Verb Phrases
    - Prepositional Phrases

Sentence Types

- Declarative Sentences
  - S → NP VP — He left
- Imperative Sentences
  - S → VP — Get out!
- Yes-No Questions
  - S → Aux NP VP — Did you decide?
- WH-Questions
  - S → WH-Word Aux NP VP — What did you decide?

Noun Phrases

- Each noun phrase has a **head noun**. -- a book
- A noun phrase the head noun may be preceded by **pre-nominal modifiers** and followed by **post-nominal modifiers**.
- Pre-Nominal Modifiers:
  - Determiner -- a, the, that, this, any, some -- a book
    - mass-nouns do not require determiners
  - Pre-Determiners -- all -- all the flights, all flights
  - Cardinal Numbers -- one, two -- two friends, one man
  - Ordinal Numbers -- first,second,next,last,other -- the last flight
  - Quantifiers -- many,several,few -- many fares
  - Adjective Phrases -- the least expensive fare
    - Adjectives can be grouped into a phrase called an **adjective phrase**.
- A simplified rule:
  - NP → (PreDet) (Det) (Card) (Ord) (Quan) (AP) NOM

Noun Phrases -- Post-Modifiers

- Three common post-modifiers:
  - prepositional phrases -- all flights from Ankara
  - non-finite clauses -- any flight arriving after 5 p.m.
    - three common non-finite post-modifiers: gerundive, -ed, and infinitive forms.
  - relative clauses -- a flight that serves dinner

NOM → NOM PP (PP) (PP)

NOM → NOM GerundVP

NOM → NOM RelClause

GerundVP → GerundV | GerundV NP | GerundV PP | GerundV NP PP

GerundV → arriving | preferring | …

RelClause → who VP | that VP

Conjunctions

- Noun phrases and other phrases can be conjoined with **conjunctions** such as *and*, *or*, *but*, …
  - *table* and *chair* ...
  - the flights that *leaving Ankara* and *arriving in Istanbul*
  - *he came from Ankara* and *he went to Istanbul*.

NP → NP and NP

VP → VP and VP

S → S and S

Recursive Structures

- Recursive rules may appear in our grammars.
  - NP → NP PP the flight from Ankara
  - VP → VP PP departed Ankara at 5 p.m.

- These rules allow us the following:
  - Flights to Ankara
  - Flights to Ankara from Istanbul
  - Flights to Ankara from Istanbul in March
  - Flights to Ankara from Istanbul in March on Friday
  - Flights to Ankara from Istanbul in March on Friday under $100
  - Flights to Ankara from Istanbul in March on Friday under $100 with lunch

Some Difficulties in Grammar Development

- When we use CFGs to describe the syntax of a natural language, we may encounter certain difficulties in the expression of some structures in natural languages.

- Some of these difficulties are:
  - Agreement
    - he flies … * he fly
    - I fly .. * I flies
    - this book * this books
    - those books * those book
  - Subcategorization
    - * I disappeared the cat. (disappear cannot be followed by a noun phrase)

Agreement

- How can we modify our grammar to handle these agreement phenomena?

- We may expand our with multiple set of rules
  - 3SgNP → …
  - Non3SgNP → …
- But this will double the size of the grammar.

- A better way to deal with agreement problems without exploding the size of the grammar by parameterizing each non-terminal with **feature structures**.

SubCategorization

- A verb phrase may consists of a verb and a number of constituents.
  - VP → Verb -- disappear
  - VP → Verb NP -- prefer a morning flight
  - VP → Verb NP PP -- leave Ankara in the morning
  - VP → Verb PP -- leaving on Monday
  - VP → Verb S -- You said there is only one flight

- Although a verb phrase can have many possible of constituents, not every verb is compatible with every verb phrase.

- Verbs have preferences for the kinds of constituents they co-occur with.
  - Transitive verbs
  - Intransitive verbs
  - Modern grammars distinguish too many subcategories (100 subcategories)

Some SubCategorization Frames

Frame	***Verb***	***Example***
Φ	eat, sleep	I want to eat
NP	prefer	I prefer a morning flight
NP NP	show	Show me all flights from Ankara
PP_{from} PP_{to}	fly	I would like to fly from Ankara to Istanbul
NP PP_{with}	help	Can you help me with a flight
VP_{to}	prefer	I would prefer to go by THY
S	mean	This means THY has a hub in Istanbul

Parsing

Parsing

- **Parsing with a CFG is the task of assigning a correct parse tree (or derivation) to a string given some grammar.**
- The correct means that it is consistent with the input and grammar.
  - It doesn't mean that it's the "right" tree in global sense of correctness.
- The leaves of the parse tree cover all and only the input, and that parse tree corresponds to a valid derivation according to the grammar.
- The parsing can be viewed as a search.
  - The search space corresponds to the space of parse trees generated by the grammar.
  - The search is guided by the structure of space and by the input.
- First, we will look at basic (bad) methods of the parsing.
  - After seeing what's wrong with them, we will look at better methods.

A Simple English Grammar

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM
NOM → NOM PP

VP → Verb
VP → Verb NP

PP → Prep NOM

Det → that | this | a | the
Noun → book | flight | meal | money
Verb → book | include | prefer

Aux → does
Prep → from | to | on

ProperNoun → Houston | TWA

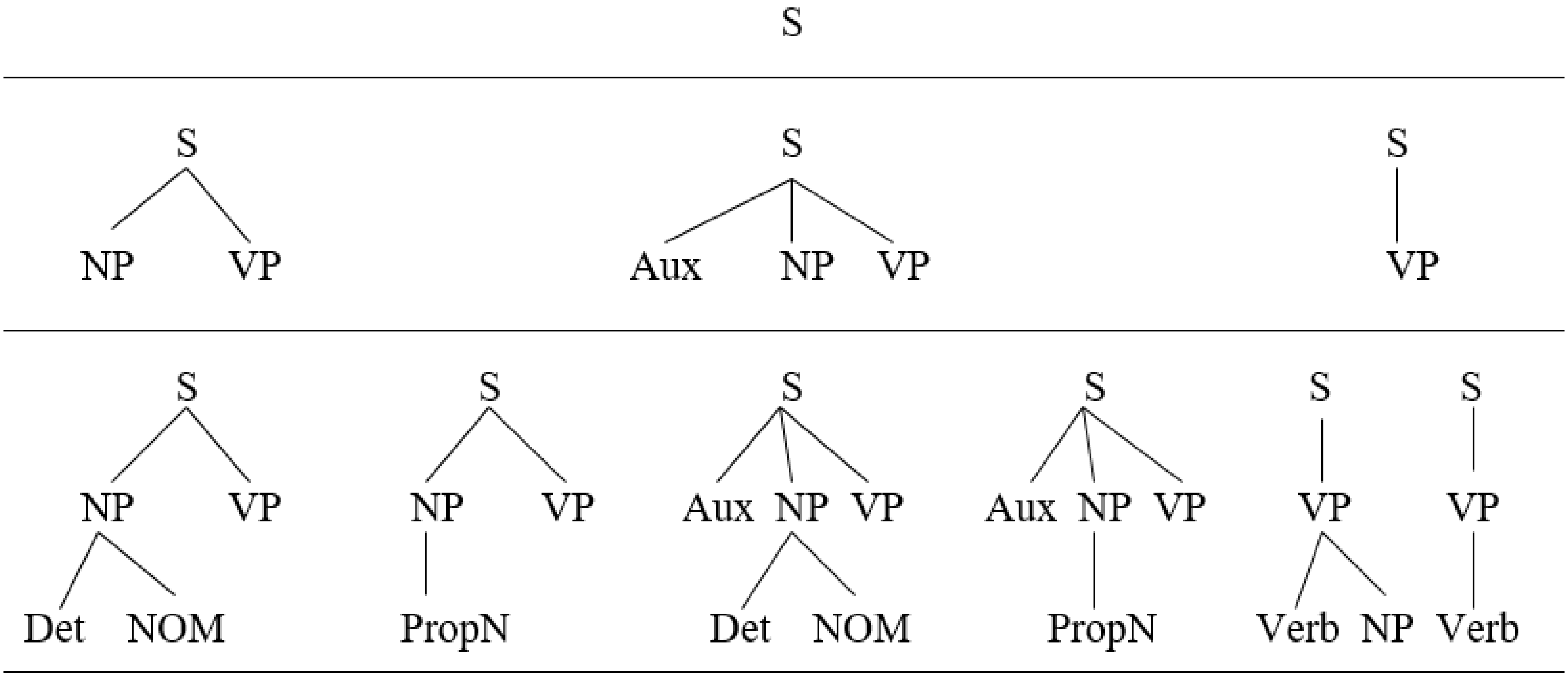
Basic Top-Down Parsing

- A top-down parser searches a parse tree by trying to build from the root node S (start symbol) down to leaves.

- First, we create the root node, then we create its children. We chose one of its children and then we create its children.

- We can search the search space of the parse trees:
  - breadth first search -- level by level search
  - depth first search -- first we search one of the children

A Top-Down Search Space



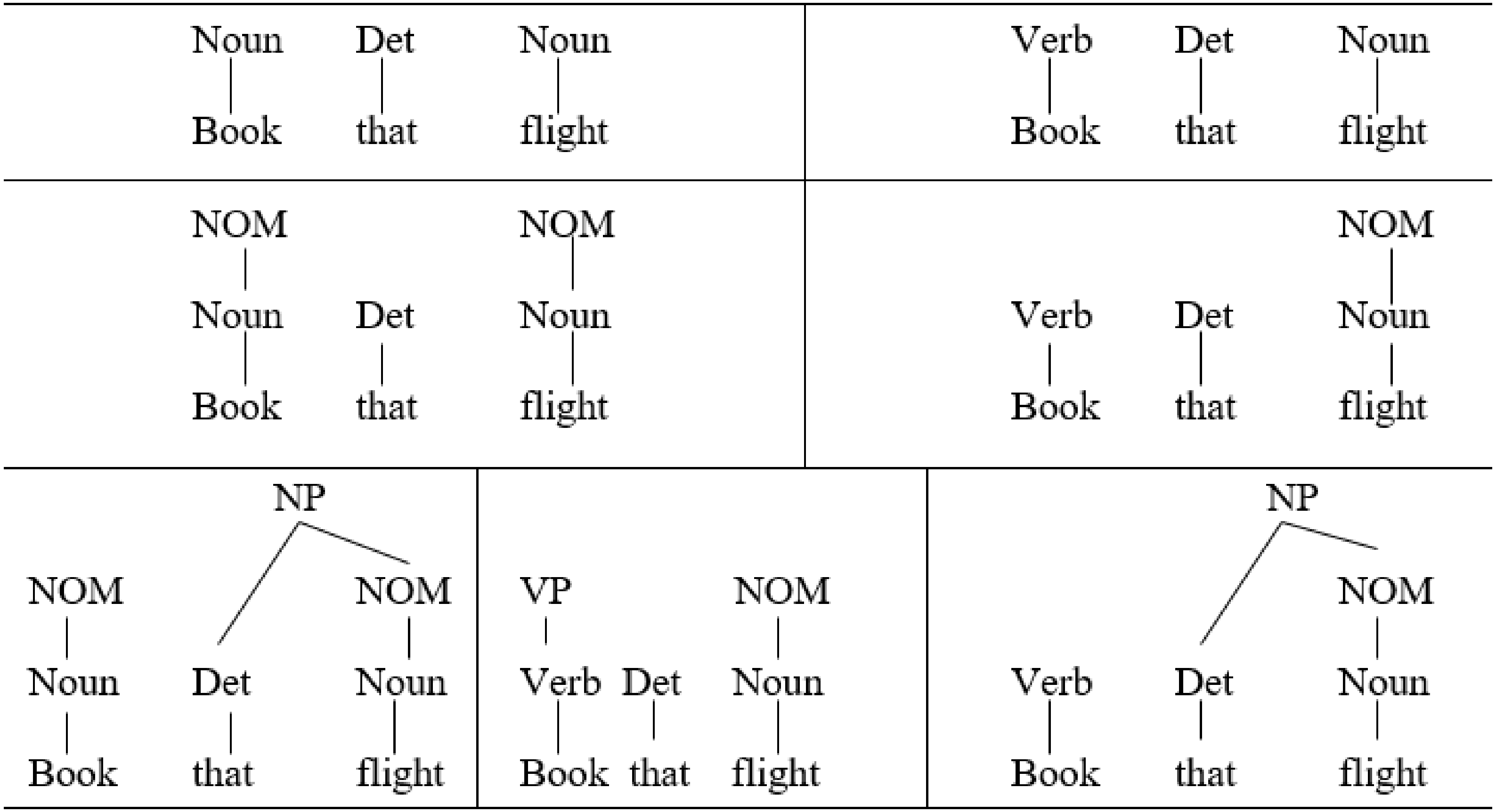
- *Input*: Book that flight

Basic Bottom-Up Parsing

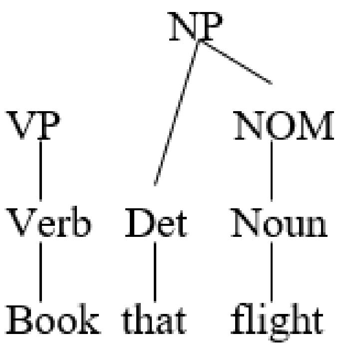
- In bottom-up parsing, the parser starts with the words of input, tries to build parse trees from words up.
- The parser is successful if the parser succeeds building a parse tree rooted in the start symbol that covers all of the input.

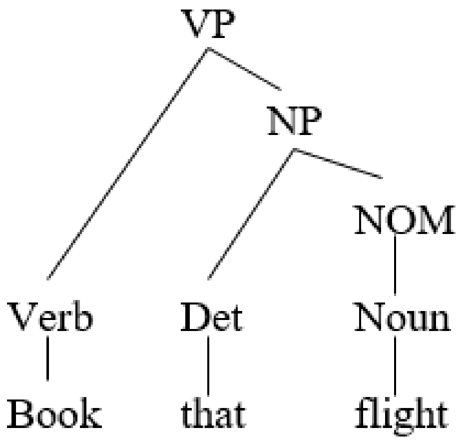
A Bottom-Up Search Space

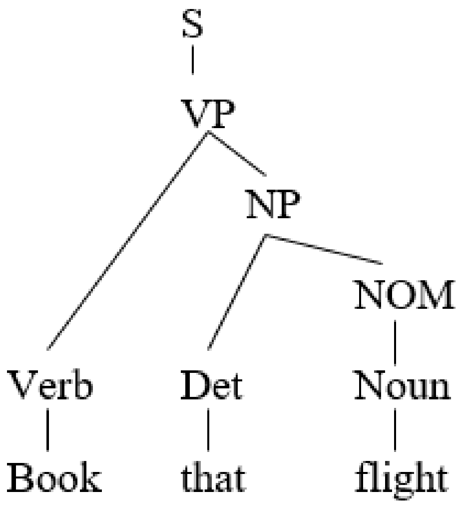
- *Input*: Book that flight



A Bottom-Up Search Space (cont.)







Top-Down or Bottom-Up?

- Each of top-down and bottom-up parsing techniques has its own advantages and disadvantages.
- The top-down strategy never wastes time exploring trees cannot result in the start symbol (starts from there).
- On the other hand, bottom-up strategy may waste time in those kind of trees.
- But the top-down strategy spends with trees which are not consistent with the input.
- On the other hand, bottom-up strategy never suggests trees that are not at least locally grounded in the actual input.
- None of these two basic strategies are good enough to be used in the parsing of natural languages.

Search Control Issues

- How our search will take place?
- Which node in the tree will be expanded next?
- Which applicable grammar rule will be tried first?

- The answers of these questions determine how to control our search in the search space of trees.
- Are we going to use depth-first or breath-first search?

A Top-Down Depth-First Left-to-Right Search

- In this top-down search, we will use:
  - depth-first strategy -- we will choose a node and explore its sub-trees
  - left-to-right -- we will choose the left-most node to explore
- For the chosen node, we will choose one of applicable rules (the first one) and we will apply it into that node.
- If there is more than one applicable rule, we keep a pointer to other applicable rules in a stack; so that if our choice fails we can backtrack to other alternatives.
- Let us look at how this method for our grammar and the following input:
  - *Does this flight include a meal?*

Top-Down Parsing with Bottom-Up Filtering

- When we choose applicable rules, we can use bottom-up information.
- For example, in our grammar we have:
  - S → NP VP
  - S → Aux NP VP
  - S → VP
- If we want to parse the input:
  - Does this flight serve a meal?
- Although all three of these rules are applicable, the first and the third ones will definitely fail because NP and VP cannot derive to strings starting with *does* (an auxiliary verb here).
- Can we make this decision before we choose an applicable rule?
  - Yes. We can use left-corner filtering.

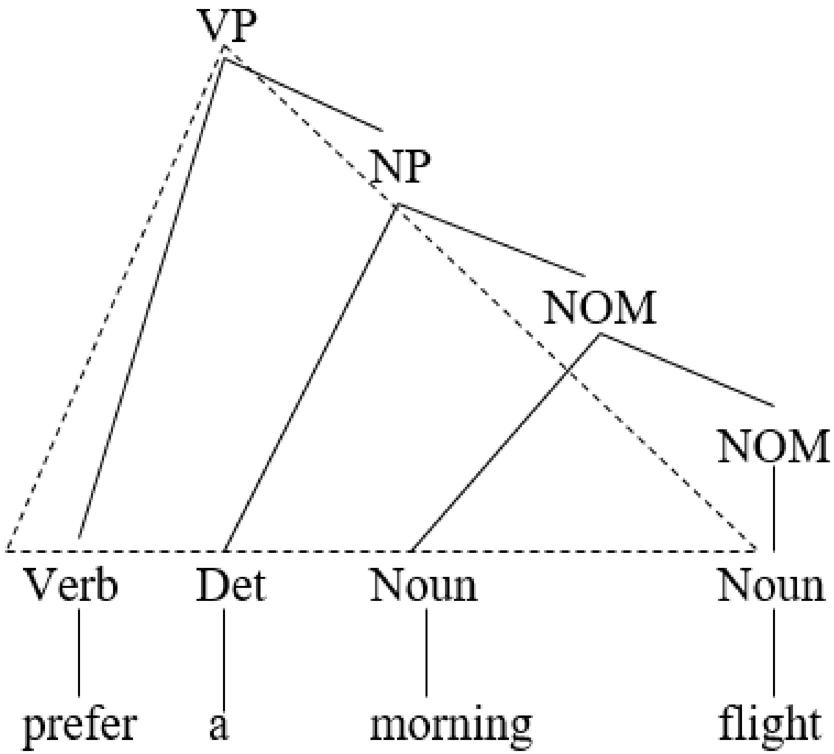
Filtering with Left Corners

- The parser should not consider any grammar rule if the current input serve as *the first word along the left edge of some derivation* from this rule.
- *The first word along the left edge of a derivation* is called as the **left-corner** of the tree.

- B is a left-corner of A if the following relation holds:
  - $A \Rightarrow^* B\alpha$
- In other words, B can be the left-corner of A if there is a derivation of A that begins with B.
- We will ask whether a part of speech (of the current input) can be left-corner of the current-node (non-terminal).

Left Corner

- *prefer* (or Verb) is a left-corner of VP



Filtering with Left-Corners (cont.)

- Do not consider any expansion where the current input can not serve as the left-corner of that expansion.

Category	*Left-Corners*
S	Det, ProperNoun, Aux, Verb
NP	Det, ProperNoun
NOM	Noun
VP	Verb
PP	Prep

Problems with Basic Top-Down Parser

- Even the top-down parser with bottom-up filtering has three problems that make it an insufficient solution to general-purpose parsing problem.
  - Left-Recursion
  - Ambiguity
  - Inefficient Reparsing of Subtrees

- First we will talk about these three problems.

- Then we will present Earley algorithm to avoid these problems.

Left-Recursion

- When left-recursive grammars are used, top-down depth-first left-to-right parsers can dive into an infinite path.
- A grammar is left-recursive if it contains at least one non-terminal A such that:
  - $A \Rightarrow^* A\alpha$
- This kind of structures are common in natural language grammars.
  - NP → NP PP
- We can convert a left-recursive grammar into an equivalent grammar which is not left-recursive.

$A \rightarrow A\beta \mid \alpha$ ==> $A \rightarrow \alpha A'$

$A' \rightarrow \beta A' \mid \varepsilon$

- Unfortunately, the resulting grammar may no longer be the most grammatically natural way to represent syntactic structures.

Ambiguity

- Top-down parser is not efficient at handling ambiguity.
- **Local ambiguity** lead to hypotheses that are locally reasonable but eventually lead nowhere. They lead to **backtracking**.
- Global ambiguity potentially leads to multiple parses for the same input (if we force it to do).
- The parsers without disambiguation tools must simply return all possible parses. But most of disambiguation tools require statistical and semantic knowledge.
- There will be many unreasonable parses. But most of applications do not want all possible parses, they want a single correct parse.
- The reason for many unreasonable parses, exponential number of parses are possible for certain inputs.

Ambiguity - Example

- If we add the following rules to our grammar:
  - VP → VP PP
  - NP → NP PP
- The following input:
  - Show me the meal on flight 286 from Ankara to Istanbul.

will have a lot of parses (14 parses?). Some of them are really strange parses.

- If we have
  - PP → Prep NP

Number of NP parses	Number of PPs
2	2
5	3
14	4
132	5
469	6

Repeated Parsing of Subtrees

- The parser often builds valid trees for portion of the input, then discards them during backtracking, only to find that it has to rebuild them again.

- The parser creates small parse trees that fail because they do not cover all the input.

- The parser backtracks to cover more input, and recreates subtrees again and again.

- The same thing is repeated more than once unnecessarily.

Repeated Parsing of Subtrees (cont.)

- Consider parsing the following NP with the following rules:

  a flight from Ankara to Istanbul on THY

  NP → Det NOM

  NP → NP PP

  NP → ProperNoun

- What happens with a top-down parser?

Repeated Parsing of Subtrees (cont.)

- *a flight from Ankara to Istanbul on THY*
- *a flight from Ankara to Istanbul on THY*
- *a flight from Ankara to Istanbul on THY*
- *a flight from Ankara to Istanbul on THY*

a flight is parsed 4 times, *from Ankara* is parsed 3 times, ...

Dynamic Programming

- We want a parsing algorithm (using dynamic programming technique) that fills a table with solutions to sub-problems that:
  - Does not do repeated work
  - Does top-down search with bottom-up filtering
  - Solves the left-recursion problem
  - Solves an exponential problem in $O(N^3)$ time.
- The answer is **Earley Algorithm**.

Earley Algorithm

- Earley Algorithm fills a table in a single pass over the input.
- The table will be size N+1 where N is the number of words in the input.
- We may think that each table entry, called state, represents gaps between words.
- Each possible subtree is represented only once, and it can be shared by all the parses that need it.

States

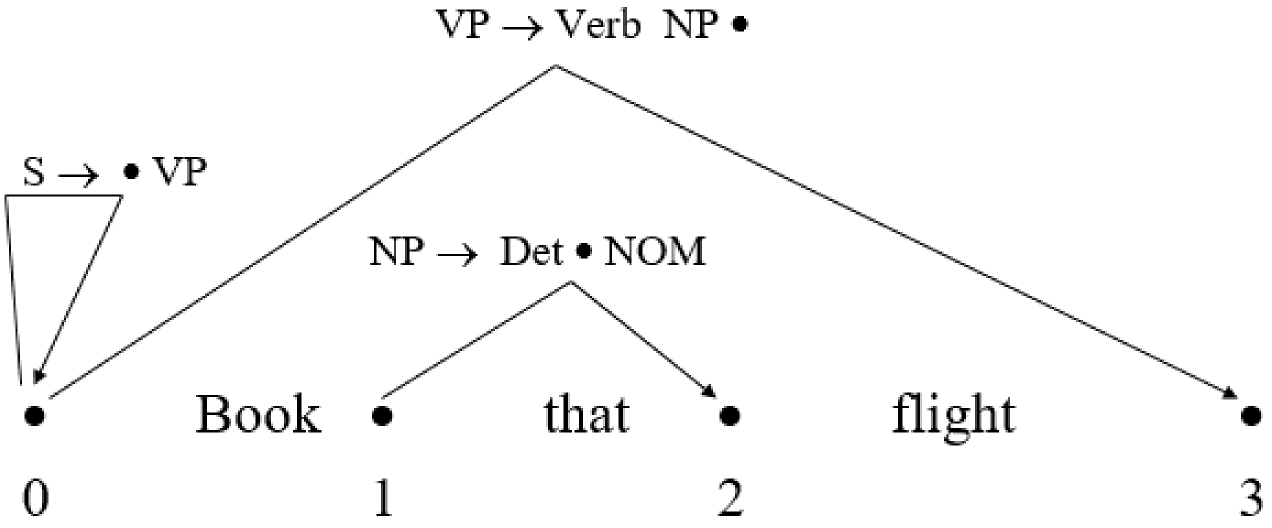
- A state in a table entry contains three kinds of information:
  - a subtree corresponding to a single grammar rule
  - information about the progress made in completing this subtree
  - the position of subtree with respect to the input.
- We use a dot in the state's grammar rule to indicate the progress made in recognizing it.
- We call this resulting structure **dotted rule**.
- A state's position are represented by two numbers indicating that where the state starts and where its dot lies.

States - Dotted Rule

- Three example states: (Ex: Book that flight)
  - $S \rightarrow \bullet VP$, [0,0]
  - $NP \rightarrow Det \bullet NOM$, [1,2]
  - $VP \rightarrow Verb\ NP \bullet$, [0,3]
- The first state represents a top-down **prediction** for S.
  - The first 0 indicates that the constituent predicted by this state should begin at position 0 (beginning of the input).
  - The second 0 indicates that the dot lies at position 0.
- The second state represents an **in-progress** constituent.
  - The constituent starts at position 1 and the dot lies at position 2.
- The third state represents a **completed** constituent.
  - This state describes that VP is successfully parsed, and that constituent covers the input from position 0 to position 3.

Graphical Representations of Dotted Rules

- A directed acyclic graph can be in the representation of dotted rules.



Parsing with Earley Algorithm

- New predicted states are based on existing table entries (predicted or in-progress) that predict a certain constituent at that spot.

- New in-progress states are created by updating older states to reflect the fact that the previously expected completed constituents have been located.

- New complete states are created when the dot in an in-progress state moves to the end.

More Specifically

1. Predict all the states

2. Read an input.
    - See what predictions you can match.
    - Extend matched states, add new predictions.
    - Go to next state (state 2)

3. At the end, see if state[N+1] contains a complete S

A Simple English Grammar (Ex.)

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM

VP → Verb
VP → Verb NP

Det → that | this | a | the
Noun → flight | meal | money
Verb → book | include | prefer

Aux → does

ProperNoun → Houston | TWA

Example: Chart[0]

book that flight

γ → • S	[0,0]	Dummy start state
S → • NP VP	[0,0]	Predictor
NP → • Det NOM	[0,0]	Predictor
NP → • ProperNoun	[0,0]	Predictor
S → • Aux NP VP	[0,0]	Predictor
S → • VP	[0,0]	Predictor
VP → • Verb	[0,0]	Predictor
VP → • Verb NP	[0,0]	Predictor

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM

VP → Verb
VP → Verb NP

Example: Chart[1]

book that flight

Verb → book •	[0,1]	Scanner
VP → Verb •	[0,1]	Completer
S → VP •	[0,1]	Completer
VP → Verb • NP	[0,1]	Completer
NP → • Det NOM	[1,1]	Predictor
NP → • ProperNoun	[1,1]	Predictor

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM

VP → Verb
VP → Verb NP

Example: Chart[2]

book that flight

Det → that •	[1,2]	Scanner
NP → Det • NOM	[1,2]	Completer
NOM → • Noun	[2,2]	Predictor
NOM → • Noun NOM	[2,2]	Predictor

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM

VP → Verb
VP → Verb NP

Example: Chart[3]

book that flight

Noun → flight •	[2,3]	Scanner
NOM → Noun •	[2,3]	Completer
NOM → Noun • NOM	[2,3]	Completer
NP → Det NOM •	[1,3]	Completer
VP → Verb NP •	[0,3]	Completer
S → VP •	[0,3]	Completer
NOM → • Noun	[3,3]	Predictor
NOM → • Noun NOM	[3,3]	Predictor

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM

VP → Verb
VP → Verb NP

Earley Algorithm

- The Earley algorithm has three main functions that do all the work.

Predictor:

- Adds predictions into the chart.
- It is activated when the dot (in a state) is in the front of a non-terminal which is not a part of speech.

Completer:

- Moves the dot to the right when new constituents are found.
- It is activated when the dot is at the end of a state.

Scanner:

- Reads the input words and enters states representing those words into the chart.
- It is activated when the dot (in a state) is in the front of a non-terminal which is a part of speech.

- The Earley algorithm uses theses functions to maintain the chart.

Predictor

```
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
```

Completer

```
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
```

Scanner

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

Enqueue

```
procedure ENQUEUE(state,chart-entry)
    if state is not already in chart-entry then
        Add state at the end of chart-entry)
    end
```

Earley Code

```
function EARLEY-PARSE(words,grammar) returns chart
  ENQUEUE(($\gamma \rightarrow \bullet$ 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 PS then
          PREDICTOR(state)
      elseif INCOMPLETE?(state) and NEXT-CAT(state) is a PS then
          SCANNER(state)
      else
          COMPLETER(state)
    end
  end
  return(chart)
```

Retrieving Parse Trees from A Chart

- To retrieve parse trees from a chart, the representation of each state must be augmented with an additional field to store information about the completed states that generated its constituents.

- To collect parse trees, we have to update COMPLETER such that it should add a pointer to the older state onto the list of previous-states of the new state.

- Then, the parse tree can be created by retrieving these list of previous-states (starting from the completed state of S).

Chart[0] - with Parse Tree Info

S0 γ → • S	[0,0]	[]	Dummy start state
S1 S → • NP VP	[0,0]	[]	Predictor
S2 NP → • Det NOM	[0,0]	[]	Predictor
S3 NP → • ProperNoun	[0,0]	[]	Predictor
S4 S → • Aux NP VP	[0,0]	[]	Predictor
S5 S → • VP	[0,0]	[]	Predictor
S6 VP → • Verb	[0,0]	[]	Predictor
S7 VP → • Verb NP	[0,0]	[]	Predictor

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM

VP → Verb
VP → Verb NP

Chart[1] - with Parse Tree Info

S8	Verb → book •	[0,1]	[]	Scanner
S9	VP → Verb •	[0,1]	[S8]	Completer
S10	S → VP •	[0,1]	[S9]	Completer
S11	VP → Verb • NP	[0,1]	[S8]	Completer
S12	NP → • Det NOM	[1,1]	[]	Predictor
S13	NP → • ProperNoun	[1,1]	[]	Predictor

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM

VP → Verb
VP → Verb NP

Chart[2] - with Parse Tree Info

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

S → NP VP
S → Aux NP VP
S → VP

NP → Det NOM
NP → ProperNoun

NOM → Noun
NOM → Noun NOM

VP → Verb
VP → Verb NP

Chart[3] - with Parse Tree Info

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

Global Ambiguity

S → Verb S → Noun

Chart[0]

S0	$\gamma \rightarrow \bullet$ S	[0,0]	[]	Dummy start state
S1	S → • Verb	[0,0]	[]	Predictor
S2	S → • Noun	[0,0]	[]	Predictor

Chart[1]

S3	Verb → book •	[0,1]	[]	Scanner
S4	Noun → book •	[0,1]	[]	Scanner
S5	S → Verb •	[0,1]	[S3]	Predictor
S6	S → Noun •	[0,1]	[S4]	Predictor