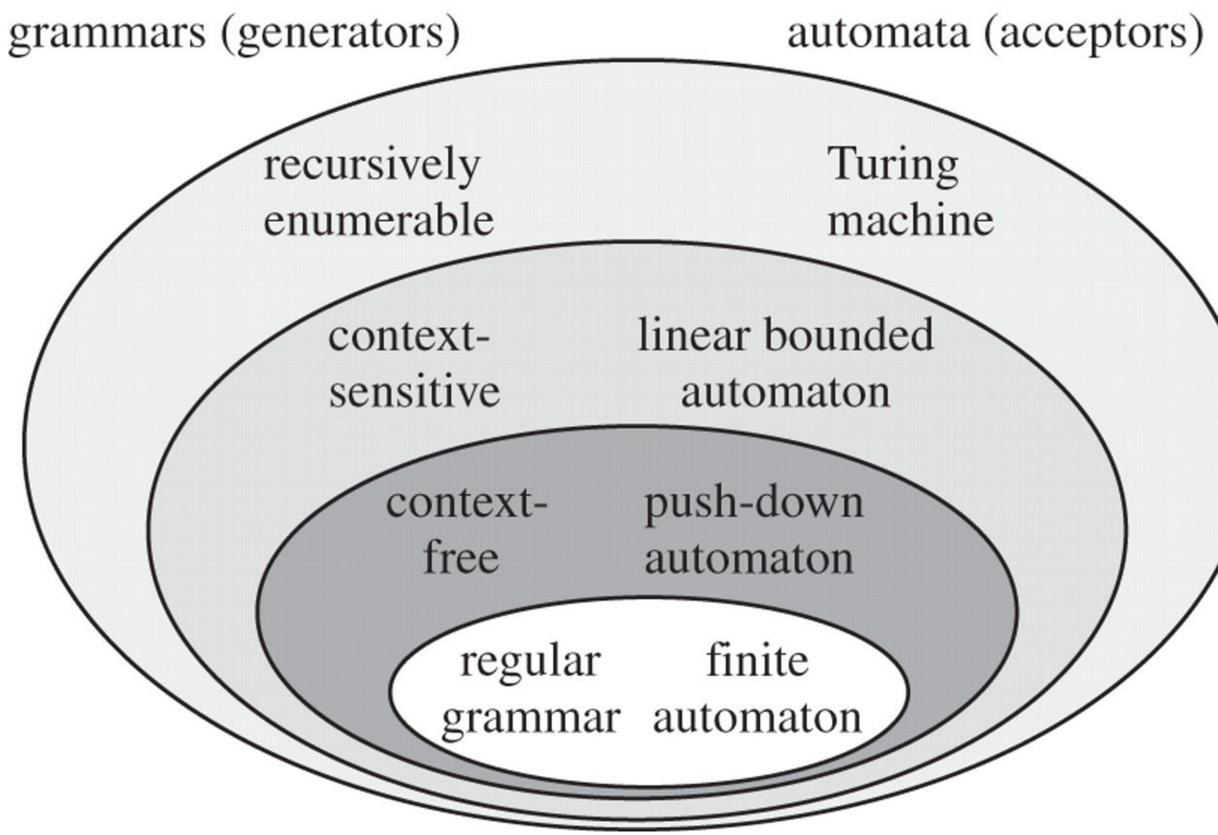
LING 473: Day 11

START THE RECORDING

Formal Grammars

Formal Grammars



Pumping lemma

- We asserted that a regular language (FSMs) cannot express $a^n b^n, n \geq 0$. How would you prove this?
- The pumping lemma for regular languages describes a property of all regular languages
  - Some other language classes have pumping lemmata too
- One way to prove that a particular language is not regular is to demonstrate that a string of the language does *not* satisfy this pumping lemma

Pumping lemma for regular languages

- Specifically, the pumping lemma says that any regular language (with an infinite number of strings) has a value $p \geq 1$ such that any string in the regular language L can be decomposed into

$$xyz, \qquad |y| \geq 1, \qquad |xy| \leq p, \qquad i \geq 0$$

such that

$$xy^i z \in L,$$

- Since there's no way to satisfy this with the string $aaabbb$, the language $a^n b^n$ is not regular.

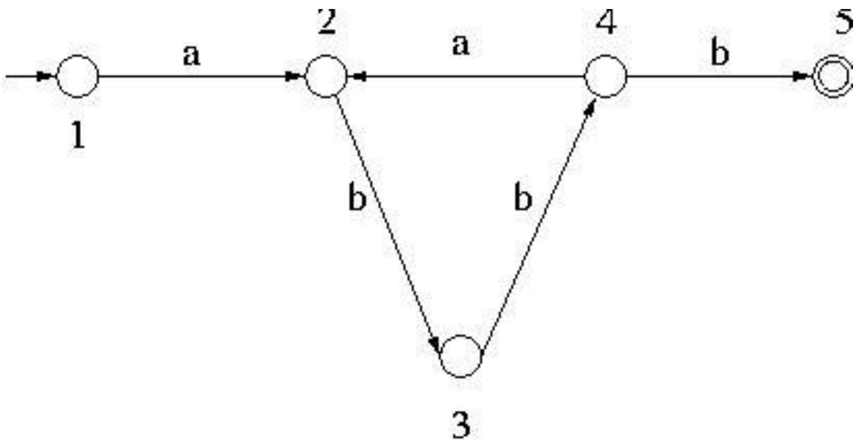
Pumping lemma

- Every string must have a non-empty “middle section” which can be repeated an arbitrary number of times, giving new strings which are all in the language
- Since there’s no way to satisfy this with the string $aaabbb$, the language $a^n b^n$ is not regular.

Pumping lemma

- For any string in a regular language, there should be a part somewhere within the first n characters that can be pumped

- Informally, this means that, if there is a loop in the automaton, you can keep going around it as many times as you like and still be generating acceptable strings



NFA accepting a(bba)*bbb

Pumping lemma

- Show that the language $a^n b^n$ is not a regular language

aaabbb

try $p = 2$

a a abbb

a aa abbb

try $p = 3$

aa a bbb

aa aa bbb

try $p = 4$

aaa b bb

aaa bb bb

$\forall$ regular languages $L, \exists\, p$:
$w = xyz, w \in L$
$|y| \geq 1, |xy| \leq p$
$\forall\, i \geq 0: xy^i z \in L$

Linear grammars

(between types 3 and 2)

- Relax regular grammars slightly to defeat that pumping lemma
- Proper superset of regular grammars (type 3)
- Proper subset of context-free grammars (type 2)

$$V = \{S\}$$
$$\Sigma = \{a, b\}$$
$$R = \{S \rightarrow aSb, S \rightarrow \varepsilon\}$$
$$S = S$$

ab, aabb, aaabbb, aaaabbbb, …

Context-free grammar

(type 2)

- What if we allow our production rule in a linear grammar to have more than one nonterminal?

…we get the most important type of grammar studied in linguistics: the context-free grammar (CFG)

Context free grammar

(type 2)

$$G = (V, \Sigma, R, S)$$

$$V = \{ \text{pre−terminals } v_0, v_1, \ldots \}$$
$$\Sigma = \{ \text{terminals } w_0, w_1, \ldots \}$$
$$R = \{ \text{rules } r_i \colon v \rightarrow \gamma \}$$
$$S = \text{start symbol}, S \in V$$

γ: sequence of terminals and pre-terminals (or Ø)

L: the language generated by G

Context-sensitive grammars

(type 1)

- No rule can make a string shorter
- Can be accepted by a 'linear bounded automaton' (LBA), a nondeterministic Turing machine which uses space $O(n)$

Unrestricted grammar

(type 0)

- The most general class
- Recognized by Turing machine
- Generate recursively-enumerable languages

Example CFG

$$G = (V, \Sigma, R, S)$$
$$V = \{\, S, NP, NOM, VP, Det, Noun, Verb, Aux \,\}$$
$$\Sigma = \{\, that, this, a, the, man, book, flight, meal, include, read, does \,\}$$
$$S = S$$

$R = \{$

S $\rightarrow$ NP VP	Det $\rightarrow$ *that* | *this* | *a* | *the*
S $\rightarrow$ Aux NP VP	Noun $\rightarrow$ *book* | *flight* | *meal* | *man*
S $\rightarrow$ VP	Verb $\rightarrow$ *book* | *include* | *read*
NP $\rightarrow$ Det NOM	Aux $\rightarrow$ *does*
NOM $\rightarrow$ Noun	
NOM $\rightarrow$ Noun NOM	
VP $\rightarrow$ Verb	
VP $\rightarrow$ Verb NP	

$\}$

Grammar rewrite rules

S --> NP VP

--> Det NOM VP

--> The NOM VP

--> The Noun VP

--> The man VP

--> The man Verb NP

--> The man read NP

--> The man read Det NOM

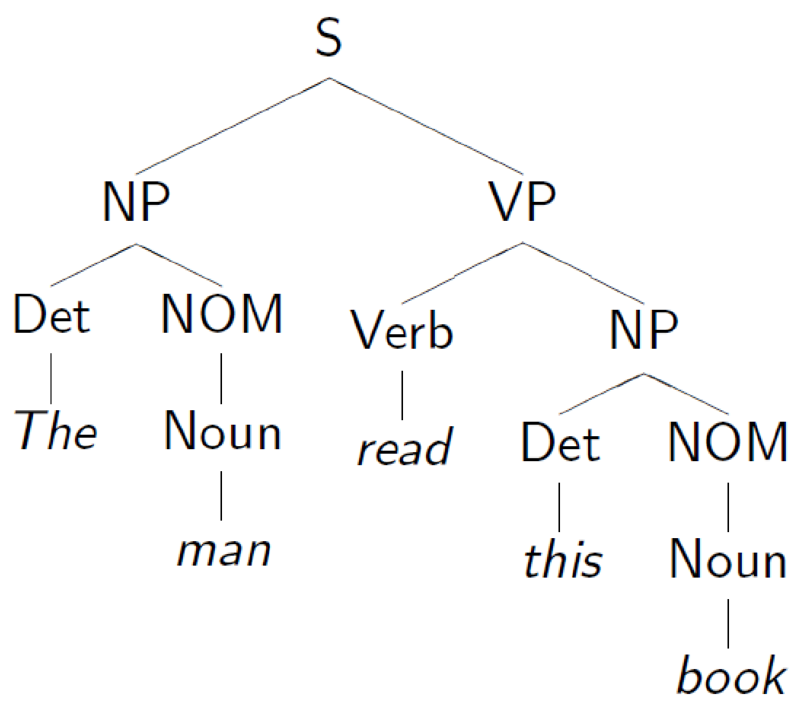
--> The man read this NOM

--> The man read this Noun

--> The man read this book

S $\longrightarrow$ NP VP	Det $\longrightarrow$ *that* | *this* | *a* | *the*
S $\longrightarrow$ Aux NP VP	Noun $\longrightarrow$ *book* | *flight* | *meal* | *man*
S $\longrightarrow$ VP	Verb $\longrightarrow$ *book* | *include* | *read*
NP $\longrightarrow$ Det NOM	Aux $\longrightarrow$ *does*
NOM $\longrightarrow$ Noun	
NOM $\longrightarrow$ Noun NOM	
VP $\longrightarrow$ Verb	
VP $\longrightarrow$ Verb NP	

Parse tree



PCFG

- Probabilistic context-free grammar
- Adds probabilities to each rule
- Each distinct left-hand-side gets a probability mass 1.0
- Rule weights can be estimated from corpora

- Why would we do this?

CFGs can express recursion

- Example of seemingly endless recursion of embedded prepositional phrases:

  PP → Prep NP

  NP → Noun PP

[S The mailman ate his [NP lunch [PP with his friend [PP from the cleaning staff [PP of the building [PP at the intersection [PP on the north end [PP of town]]]]]]].

Most programming languages are type-2 grammars (CFGs)

Chomsky Normal Form

- Any CFG can be converted into a form where all rules are of the form:

$$X \rightarrow YZ$$
$$X \rightarrow a$$
$$S \rightarrow \lambda$$

(S is the only terminal that can go to the empty string)

Convert $W \rightarrow X\ Y a\ Z$ to Chomsky Normal Form

CNF conversion

- Steps:
  1. Make S non-recursive
  2. Eliminate λ (except $S \to \lambda$)
  3. Eliminate all chain rules
  4. Remove unused symbols

Convert the following grammar to Chomsky Normal Form:

$$S \to ASA$$
$$S \to aB$$
$$A \to B$$
$$A \to S$$
$$B \to b$$
$$B \to \lambda$$

CNF conversion

$S \rightarrow ASA$
$S \rightarrow aB$
$A \rightarrow B$
$A \rightarrow S$
$B \rightarrow b$
$B \rightarrow \lambda$

$S \rightarrow ASA$
$S \rightarrow U_a B$
$A \rightarrow B$
$A \rightarrow S$
$B \rightarrow b$
$B \rightarrow \lambda$
$U_a \rightarrow a$

$S \rightarrow AX$
$S \rightarrow U_a B$
$A \rightarrow B$
$A \rightarrow S$
$B \rightarrow b$
$B \rightarrow \lambda$
$U_a \rightarrow a$
$X \rightarrow SA$

$S \rightarrow S'$
$S' \rightarrow AX$
$S' \rightarrow U_a B$
$A \rightarrow B$
$A \rightarrow S'$
$B \rightarrow b$
$B \rightarrow \lambda$
$U_a \rightarrow a$
$X \rightarrow S'A$

$S \rightarrow S'$
$S' \rightarrow AX$
$S' \rightarrow U_a B$
$A \rightarrow B$
$A \rightarrow S'$
$B \rightarrow b$

$U_a \rightarrow a$
$X \rightarrow S'A$
$X \rightarrow S'$
$S' \rightarrow X$
$S' \rightarrow U_a$

$$\begin{array}{l} S \to S' \\ S' \to AX \\ S' \to U_a B \\ A \to B \\ A \to S' \\ B \to b \\ U_a \to a \\ X \to S'A \\ X \to S' \\ S' \to X \\ S' \to U_a \end{array}$$

$$\begin{array}{l} S \to X \\ X \to AX \\ X \to U_a B \\ A \to B \\ A \to X \\ B \to b \\ U_a \to a \\ X \to XA \\ \cancel{X \to S'} \\ \cancel{S' \to X} \\ S \to a \end{array}$$

$$\begin{array}{l} \cancel{S \to X} \\ X \to AX \\ X \to U_a B \\ A \to B \\ A \to X \\ B \to b \\ U_a \to a \\ X \to XA \\ S \to AX \\ S \to U_a B \\ S \to XA \\ S \to a \end{array}$$

$$\begin{array}{l} X \to AX \\ X \to U_a B \\ A \to b \\ A \to X \\ B \to b \\ U_a \to a \\ X \to XA \\ S \to AX \\ S \to U_a B \\ S \to XA \\ S \to a \end{array}$$

all done

$$\begin{array}{l} X \to AX \\ X \to U_a B \\ A \to b \\ \cancel{A \to X} \\ B \to b \\ U_a \to a \\ X \to XA \\ S \to AX \\ S \to U_a B \\ S \to XA \\ A \to AX \\ A \to U_a B \\ A \to XA \\ S \to a \end{array}$$

Example: You Try It!

Given the language L:

$S \rightarrow AbA$
$A \rightarrow Aa$
$A \rightarrow \lambda$

Convert to Chomsky normal form.

Step One: Let S be non-recursive

Done for us!

$$S \rightarrow AbA$$
$$A \rightarrow Aa$$
$$A \rightarrow \lambda$$

Step 2: Only S can be empty

Solution: rewrite empty pre-terminals and terminals by decomposing into many rules

$$S \rightarrow AbA$$
$$A \rightarrow Aa$$
$$A \rightarrow \lambda$$

$$S \rightarrow AbA$$
$$S \rightarrow Ab$$
$$S \rightarrow bA$$
$$S \rightarrow b$$
$$A \rightarrow Aa$$
$$A \rightarrow a$$

Step 3: Decompose triples

$$S \rightarrow AbA$$
$$S \rightarrow Ab$$
$$S \rightarrow bA$$
$$S \rightarrow b$$
$$A \rightarrow Aa$$
$$A \rightarrow a$$

$$S \rightarrow ZA$$
$$S \rightarrow Ab$$
$$S \rightarrow bA$$
$$S \rightarrow b$$
$$A \rightarrow Aa$$
$$A \rightarrow a$$
$$Z \rightarrow Ab$$

Step 4: Segregate Pre-terminals and Terminal

$$S \rightarrow ZA$$
$$S \rightarrow Ab$$
$$S \rightarrow bA$$
$$S \rightarrow b$$
$$A \rightarrow Aa$$
$$A \rightarrow a$$
$$Z \rightarrow Ab$$

$$S \rightarrow ZA$$
$$S \rightarrow AB$$
$$S \rightarrow BA$$
$$S \rightarrow B$$
$$A \rightarrow AY$$
$$A \rightarrow a$$
$$Z \rightarrow AB$$
$$B \rightarrow b$$
$$Y \rightarrow a$$

Parsing context-free grammars (type 2)

- CFGs are widely used to represent surface syntax in natural languages
- Space complexity:
  - you'll need at least one stack
  - space use will depend on the amount of recursion in the input
- Time complexity
  - Generally $O(n^3)$

Parsing

- The opposite of generation: find the structure from a string
- Essentially a search problem
- Find all structure that match an input string
- Two approaches
  - Bottom-up
  - Top-down

Recognizer v. parser

- Recognizer (acceptor) is a program that determines whether a sentence is accepted by the grammar or not
- A parser determines this as well, and if the sentence is accepted, it also returns the structural configuration(s) of grammar rules for the sentence
- Some parsing systems may also produce compositional semantics

Soundness and completeness

- Correctness: a parser is sound if every parse it returns is correct
- A parser terminates if it is guaranteed not to enter an infinite loop
- A parser is complete for grammar G and sentence S if it is sound, produces every possible parse for S, and terminates
- Often, we settle for sound but incomplete parsers
  - probabilistic parsers may be able to return the k-best parses

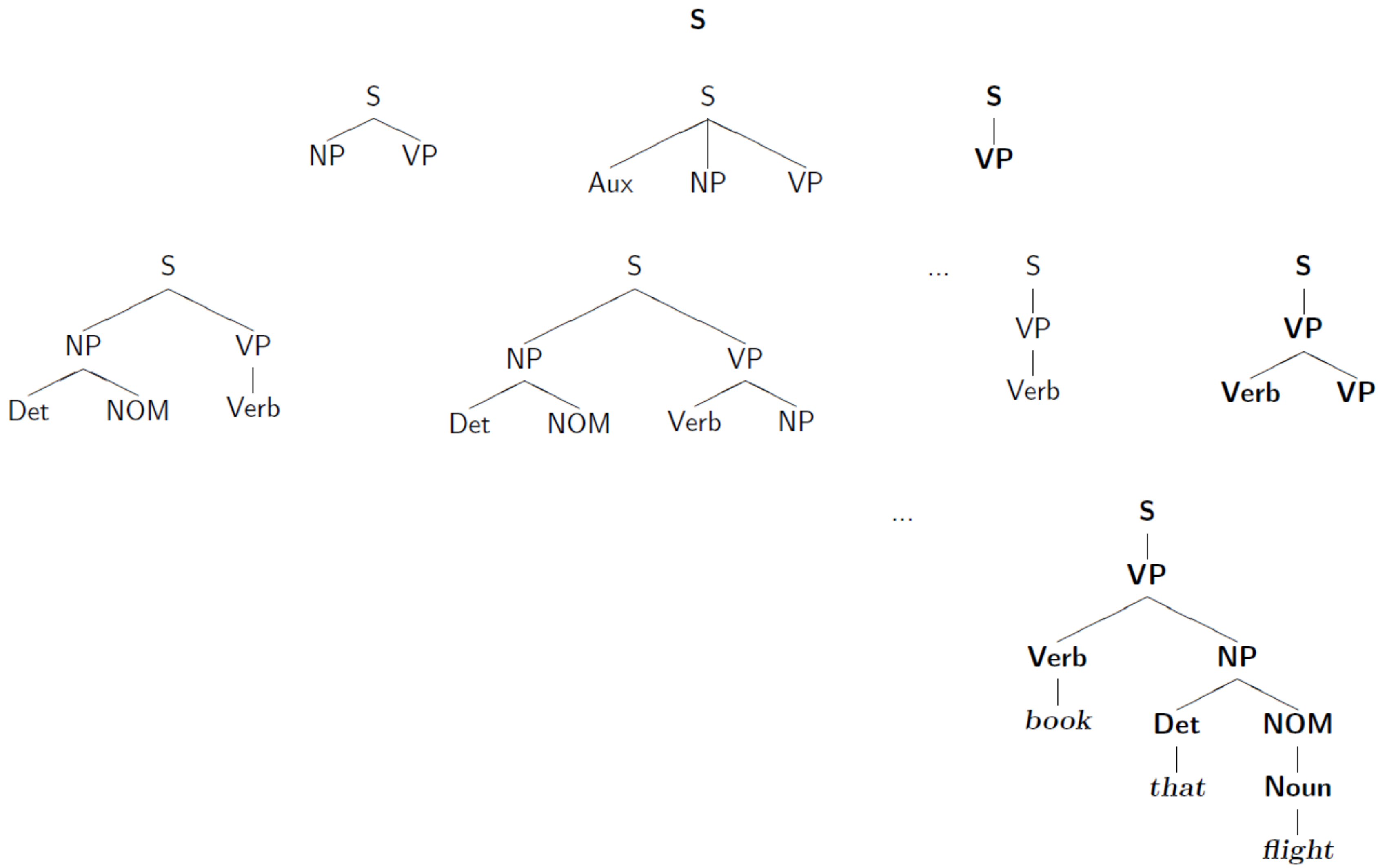
Top-down parsing

- Create a list of goal constituents
- Rewrite goals by matching a goal on the list with the left-hand-side of a rule
- Replace with the right-hand-side
- If there is more than one match to the LHS, try different rules (breadth-first or depth-first)

example

S $\longrightarrow$ NP VP	Det $\longrightarrow$ *that* | *this* | *a* | *the*
S $\longrightarrow$ Aux NP VP	Noun $\longrightarrow$ *book* | *flight* | *meal* | *man*
S $\longrightarrow$ VP	Verb $\longrightarrow$ *book* | *include* | *read*
NP $\longrightarrow$ Det NOM	Aux $\longrightarrow$ *does*
NOM $\longrightarrow$ Noun	
NOM $\longrightarrow$ Noun NOM	
VP $\longrightarrow$ Verb	
VP $\longrightarrow$ Verb NP	

Book that flight.


S
S
NP VP
S
Aux NP VP
S
VP
S
NP VP
Det NOM Verb
S
NP VP
Det NOM Verb NP
...
S
VP
Verb
S
VP
Verb VP
...
S
VP
Verb NP
book Det NOM
that Noun
flight

Problems with top-down parsing

- Left-recursive rules lead to infinite recursion

$$NP \rightarrow NP\ PP$$

- Poor performance when there are many matches for an LHS
  - If there are many rules for S, there's no way to eliminate irrelevant ones. In other words, it does the useless work of expanding things that there is no evidence for
- Doesn't work at the terminals (lexemes)
- Can't make use of common substructure

Bottom-up parsing

- Bottom-up parsing is data-directed
- Start with the string to be parsed
- Match right-hand-sides, condense to LHS
  - Still need to choose when there are multiple possible matches for the RHS
  - Can use breadth-first or depth-first search
- Parsing is complete when all you have left is the start symbol

example

S $\longrightarrow$ NP VP	Det $\longrightarrow$ *that* | *this* | *a* | *the*
S $\longrightarrow$ Aux NP VP	Noun $\longrightarrow$ *book* | *flight* | *meal* | *man*
S $\longrightarrow$ VP	Verb $\longrightarrow$ *book* | *include* | *read*
NP $\longrightarrow$ Det NOM	Aux $\longrightarrow$ *does*
NOM $\longrightarrow$ Noun	
NOM $\longrightarrow$ Noun NOM	
VP $\longrightarrow$ Verb	
VP $\longrightarrow$ Verb NP	

Book that flight.

Shift-reduce parsing

Stack	Input remaining	Action
()	Book that flight	shift
(Book)	that flight	reduce, Verb $\rightarrow$ book, (Choice #1 of 2)
(Verb)	that flight	shift
(Verb that)	flight	reduce, Det $\rightarrow$ that
(Verb Det)	flight	shift
(Verb Det flight)		reduce, Noun $\rightarrow$ flight
(Verb Det Noun)		reduce, NOM $\rightarrow$ Noun
(Verb Det NOM)		reduce, NP $\rightarrow$ Det NOM
(Verb NP)		reduce, VP $\rightarrow$ Verb NP
(Verb)		reduce, S $\rightarrow$ V
(S)		SUCCESS!

Ambiguity may lead to the need for backtracking.

Shift-reduce parser

- Start with the sentence in an input buffer
  - Shift: push the next input symbol onto the stack
  - Reduce: if a RHS matches the top elements of the stack, pop those elements off and push the LHS
- If either shift or reduce are possible, choose arbitrarily
- If you end up with only the start symbol on the stack, you have a parse
- Otherwise, you can backtrack

Shift-reduce parser

- In the top-down parser, the main decision was which production rule to pick
- In a bottom-up shift-reduce parser, the decisions are:
  - Should we shift, or reduce
  - If we reduce, then by which rule

  Both of these decisions can be revisited when backtracking

Problems with bottom-up parsing

- No obvious way to generate structures that generate empty surface elements
- Lexical ambiguity can explode the search space
- Useless constituents can be built locally

Top-down and bottom-up parsers can both be extremely inefficient on real-world NLP parsing problems. Complexity may approach $O(k^n)$ in the sentence length.

Parsing is hard

- Left-recursive structures must be found, not predicted
- Empty categories must be predicted, not found
- When backtracking, don’t redo any work

Next time

- Clustering
- Classifiers
- Overview of Information Theory