Lecture 10

August 23, 2016

Formal grammars

Reminder: start the recording

Announcements

- Project 4: DNA
  - Due at 11:45 p.m. on 9/1 (one week from Thursday)
- Project 5: Naïve Bayesian language classifier
  - Due at 11:45 p.m. on Thursday Sept. 8th

  http://courses.washington.edu/ling473/Project5.pdf
- Reminder: Writing Assignment

  http://courses.washington.edu/ling473/writing-assignment.html
  - Due at 11:45 p.m. on Tuesday, Sept. 6th

Self-quiz answers

Two fair coins are tossed. One of them shows heads and the other rolls under the couch. What is that chance that the hidden coin is showing tails?

There is much controversy surrounding the frequentist vs. Bayesian interpretation of this type of problem

- http://en.wikipedia.org/wiki/Marilyn_vos_Savant
- http://en.wikipedia.org/wiki/Boy_or_Girl_paradox

In the non-frequentist approach, the answer would be $\frac{2}{3}$

(H,H) (H,T) (T,H) (T,T)

One head is showing already

(H,H) (H,T) (T,H) ~~(T,T)~~

1 ✗ 2 ✓ 3 ✓

2 out of 3 remaining possibilities have a hidden ‘tails’

In[1]:= □ □ **Tuples** , ***T*** ,

Out[1]= H, H , H, T , T, H , T, T

Size of the sample space:

In[2]:= ***n*** □**Length** □

Out[2]= 4

Assuming heads and tails are equally likely, the unconditional probability of one head and one tail:

In[4]:= $\mathbf{Pr}\ \boldsymbol{H}\ \square \boldsymbol{T} \square \mathbf{1}\ \square\ \dfrac{\textbf{Length Select } \square\textbf{, MemberQ } \square\textbf{1, } \boldsymbol{H}\quad \textbf{MemberQ } \square\textbf{1, } \boldsymbol{T}\ \textbf{\&}}{\boldsymbol{n}}$

Out[4]= $\frac{1}{2}$

In[7]:= $\mathbf{Pr}\ \boldsymbol{T}\ \square\ \mathbf{1}\ \square\ \dfrac{\textbf{Length Select } \square\textbf{, MemberQ } \square\textbf{1, } \boldsymbol{T}\ \textbf{\&}}{\boldsymbol{n}}$

Out[7]= $\frac{3}{4}$

Probability of exactly one head and one tail given that one is head:

In[8]:= $\mathbf{Pr}\ \boldsymbol{H}\ \square \boldsymbol{T} \square \mathbf{1} \boldsymbol{H}\ \square\ \mathbf{1}\qquad \dfrac{\mathbf{Pr}\ \boldsymbol{H}\ \square\ \boldsymbol{T}\ \square\ \mathbf{1}}{\mathbf{Pr}\ \boldsymbol{T}\ \square\ \mathbf{1}}$

Out[8]= $\frac{2}{3}$

There are two boxes. One contains two black marbles and two white marbles. The other contains four black marbles and one white marble. In this experiment, a box will be selected randomly, and then a marble will be drawn randomly from the selected box.

- What is the probability that the marble will be black?
- We run the experiment and it turns out that the selected marble is white. What is the probability that the first box was selected?

In[19]:= **B1** □ ***b b, w, w***

In[20]:= **B2** □ ***b b, b, b, w***

Assuming the selection of either of the two boxes is equally likely, we record the prior probability that the first box was selected (and complement) from the problem statement:

In[21]:= $\mathbf{Pr\ FS} \ \square \ \mathbf{Pr\ FS}^{C} \ \square \ \frac{\mathbf{1}}{\mathbf{2}}$

Out[21]=

$\frac{1}{2}$

Conditional probabilities of drawing black given the box selection:

In[22]:= $\mathbf{Pr\ DB\ FS} \ \square \ \frac{\mathbf{Count\ B1}, \boldsymbol{b}}{\mathbf{Length\ B1}}$

Out[22]=

$\frac{1}{2}$

In[23]:= $\mathbf{Pr\ DB\ \ FS}^{C} \ \square \frac{\mathbf{Count\ B2}, \boldsymbol{b}}{\mathbf{Length\ B2}}$

Out[23]=

$\frac{4}{5}$

Overall prior probability of drawing black:

In[33]:= $\mathbf{Pr\ DB} \ \square \ \ \mathbf{Pr\ FS\ Pr\ DB\ FS} \ \square \ \mathbf{Pr\ FS}^{C}\ \mathbf{Pr\ DB\ FS}^{C}$

Out[33]=

$\frac{13}{20}$

In[34]:= **N** □

Out[34]=

0.65

For part (b.), use Bayes' theorem

In[35]:= **Pr DW □ 1 □ Pr DB**

Out[35]=

$$\frac{7}{20}$$

In[36]:= **Pr DW FS □** $\frac{\textbf{Count B1, } w}{\textbf{Length}}$

Out[36]=

$$\frac{1}{2}$$

Now it is a simple matter to calculate the probability that the first was previously selected, given the observation of a white marble:

In[37]:= **Pr FS DW □** $\frac{\textbf{Pr DW FS Pr FS}}{\textbf{Pr DW}}$

Out[37]=

$$\frac{5}{7} = 0.714286$$

The high school Shakespeare club has 5 freshman boys, 7 freshman girls, and 6 sophomore boys. How many sophomore girls must be in the club in order for gender and class to be independent when a student is chosen at random from the club?

Let n be the number of sophomore girls

In[5]:= $\mathbf{Pr[boy, freshman]} = \dfrac{\mathbf{5}}{\boldsymbol{n}+\mathbf{18}}$

In[6]:= $\mathbf{Pr[boy]} = \dfrac{\mathbf{11}}{\boldsymbol{n}+\mathbf{18}}$

In[7]:= $\mathbf{Pr[freshman]} = \dfrac{\mathbf{12}}{\boldsymbol{n}+\mathbf{18}}$

In[8]:= **Solve[Pr[boy, freshman] == Pr[boy] Pr[freshman], *n*]**

Out[8]= $n \to \dfrac{42}{5}$ = 8.4

It is ***not possible*** for gender and class to be independent

The linguistics section at the library has three books on Austronesian languages. We choose two linguistics books at random. The probability of them both being on Austronesian language is $\frac{1}{1650} = 0.000606061$. How many linguistics books are there?

In[185]:= **Replace** n **Solve** $\left(\frac{3}{n}\right)\left(\frac{2}{n □ 1}\right)$ □ **.000606061,** n

Out[185]= □99., 100.

Discard a negative solution:

In[186]:= **Select** □ , □ □ **0 &**

Out[186]= 100.

A multiple choice exam is given. A problem has four possible answers, and exactly one answer is correct. The student is allowed to select as many answers as he likes. If his chosen subset contains the correct answer, the student receives three points, but he loses one point for each wrong answer in his chosen subset. What is the expected score?

zero.

- A die is rolled TT times until 6 is shown. What is the probability distribution for TT?

- *Geometric* with $pp = \frac{1}{6}$
- Expected value: 6

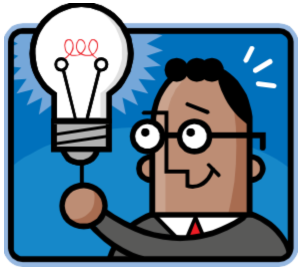
Note: this would be the same answer for 1, 2, 3, 4, 5…

A die is rolled TT times until 6 is shown. Let EE denote the event that $TT > 3$. Let FF denote the event that $TT > 6$. Calculate $PP(EE)$ and $PP(FF|EE)$.

- from last week's lecture: for a geometric distribution, $PP(XX > xx) = (1 - pp)^{xx}$

$$\left(1 - \frac{1}{6}\right)^3 = \frac{125}{216} = .5787$$

7c. alternate



$$
\begin{aligned}
& PP(TT > 3) \\
& = 1 - PP(TT \leq 3) \\
& = 1 - \left(PP(TT = 1) + PP(TT = 2) + PP(TT = 3)\right) \\
& = 1 - \sum_{xx=1}^{3} (1 - pp)^{xx-1} pp \\
& = \frac{125}{216} \\
& = .5787
\end{aligned}
$$

7d.

$$
\begin{aligned}
P(F|E) &= \\
&= P(T > 6 | T > 3) \\
&= \frac{P(T > 6, T > 3)}{P(T > 3)} \\
&= \frac{P(T > 6)}{P(T > 3)} \\
&= \frac{(1-p)^6}{(1-p)^3} \\
&= \frac{125}{216}
\end{aligned}
$$

In other words, given that you've rolled the die 3 times and you have not gotten a 6 yet, what is the probability that you won't get a 6 within the next three rolls?

Notice: the probability hasn't changed, suggesting that the trials (each roll) are independent.

Tagging objective function

Predict a sequence of tags t based on the probability of tags and words $P(t_i | w_i)$. Given sentence

“t-hat”

$$S = (w_0, w_1, \ldots w_n)$$
$$t = \mathbf{argmax}_{t_i} P(t_i | w_i).$$

“t is the best sequence of tags that match a tag t_i to its word w_i.”

This material is also covered in section 5.5 (p.139) of Jurafsky & Martin, 2nd ed.

Simplistic tagger

$$S = (ww_0, ww_1, \ldots ww_{nn})$$
$$tt = \mathrm{argmax}_{tt_{ii}} PP(tt_{ii} \,|\, ww_{ii})$$

repeated from last slide

This is surely the function we want to maximize, but it’s not clear how to calculate the probabilities $PP(tt|ww)$.

Simplistic tagger: Why don’t we use probabilities calculated from a corpus ?

– like you did for Assignment 3

Simplistic tagger

DT	NN	VBD	RB	IN	DT	NN	,	CC	DT	VBG	NNS	VBD	DT	NN	VBD
the	cold	passed	reluctantly	from	the	earth	,	and	the	retiring	fogs	revealed	an	army	stretched

IN	IN	DT	NNS	,	VBG	.	IN	DT	NN	VBN	IN	JJ	TO	VB	,	DT	NN	VBN
out	on	the	hills	,	resting	.	as	the	landscape	changed	from	brown	to	green	,	the	army	awakened

,	CC	VBD	TO	VB	IN	NN	IN	DT	NN	IN	NNS	.
,	and	began	to	tremble	with	eagerness	at	the	noise	of	rumors	.

$$\text{argmax}_{t} P(t|\text{the}) = \text{DT}$$

$$\text{argmax}_{t} P(t|\text{cold}) = \text{JJ}$$

How well does the simplistic tagger work?

- Such a POS tagger is not really usable

Most probable POS tag: JJ
Correct tag: NN

Most probable POS tag: VBG
Correct tag: JJ

DT		VBD	RB	IN	DT	NN	,	CC	DT		NNS	VBD	DT	NN	VBD
the	cold	passed	reluctantly	from	the	earth	,	and	the	retiring	fogs	revealed	an	army	stretched

IN	IN	DT	NNS	,	VBG	.	IN	DT	NN	VBN	IN	JJ	TO	JJ	,	DT	NN	VBN
out	on	the	hills	,	resting	.	as	the	landscape	changed	from	brown	to	green	,	the	army	awakened

,	CC	VBD	TO	VB	IN	NN	IN	DT	NN	IN	NNS	.
,	and	began	to	tremble	with	eagerness	at	the	noise	of	rumors	.

Use Bayes Theorem

Of course, you have this memorized

$$PP(AA|BB) = \frac{PP(BB|AA)PP(AA)}{PP(BB)}$$

Remember, this was our objective function

$$tt = \mathrm{argmax}_{tt} PP(tt_{ii} | ww_{ii})$$

$$tt = \mathrm{argmax}_{tt} \frac{PP(ww_{ii}|tt_{ii})PP(tt_{ii})}{PP(ww_{ii})}$$

This is one of the most important slides of this entire class

For each evaluated value of ii, $PP(ww_{ii})$ will be the same. We can cancel it.

$$tt = \text{argmax}_{tt} \frac{PP(ww_{ii}|tt_{ii})PP(tt_{ii})}{\cancel{PP(ww_{ii})}}$$

$$tt = \text{argmax}_{tt} PP(ww_{ii}|tt_{ii})PP(tt_{ii})$$

The best sequence of tags is determined by the probability of each word given its tag and also the probability of that tag.

I repeat…

$$t = \text{argmax}_t P(w_i | t_i) P(t_i)$$

“likelihood”

“prior”

“We compute the most probable tag sequence… by multiplying the **likelihood** and the **prior probability** for each tag sequence and choosing the tag sequence for which this product is greatest.

“Unfortunately, this is still too hard to compute directly…”

Jurafsky & Martin (paraphrase) p.140

We still need to make some assumptions.

we'll come back to this part later

$$t = \text{argmax}_t P(w_i|t_i) P(t_i)$$

Assumption 1: If we want to use corpus probabilities to estimate $P(w_i|t_i)$, we need to formally note that we're assuming

$$P'(w_i|t_i) \approx \underset{i}{?} P(w_i|t_i)$$

"The only POS tag a word depends on is its own."

Any progress?

- So wait: if we're assuming the only *POS tag* a *word* depends on is its own, how is this going to be better than the simplistic tagger from before, which assumed that the only *word* a *POS tag* depends on is its own?
- In other words, Why is $PP(ww|tt)$ going to work better than $PP(tt|ww)$?
- Hint: $|\Omega|$
- Hint: $|\mathrm{T}| \ll |\mathrm{W}|$

Answer: because there are a lot more distinct words than tags, conditioning on *tags* rather than *words* increases the resolution of the corpus measurements

example

$$P(\text{cold}|\text{NN}) = .00002$$
$$P(\text{cold}|\text{JJ}) = .00040$$

$$P(\text{JJ}|\text{cold}) = .97$$
$$P(\text{NN}|\text{cold}) = .03$$

This value will drown out our calculation and we’d never tag “cold” as a noun!

$$t = \text{argmax}_t P'(w_i | t_i) P(t_i)$$

Assumption 2: The only tags that a tag t_i depends on are the n previous tags, $t_{i-n-1} \ldots t_{i-1}$. For example, in a POS bigram model:

$$P'(t_i) \approx \prod_i P(t_i | t_{i-1})$$

This is known as the bigram assumption: "The only POS tag(s) a POS tag depends on are the ones immediately preceding it."

Putting it together

$$tt = \text{argmax}_{tt} PP(ww_{ii}|tt_{ii})PP(tt_{ii})$$

$$PP'(ww_{ii}|tt_{ii}) \approx ? \; PP(\;|\;)$$

$ww_{ii}\ tt_{ii}$

ii

$$PP'(tt_{ii}) \approx ? \; PP(\;|tt_{ii}\ tt_{ii-1})$$

ii

$$tt = \text{argmax}_{tt} \; ? \; PP(ww_{ii}|tt_{ii}) \; ? \; PP(tt_{ii}|tt_{ii-1})$$

ii ii

$$tt = \text{argmax}_{tt} \; ? \; PP(ww_{ii}|tt_{ii})\,PP(tt_{ii}|tt_{ii-1})$$

ii

Reminder: estimating $P(w_i|t_i)$ from a corpus

Definition of conditional probability

$$P(A|B) = \frac{P(A,B)}{P(B)}$$

$$P(A|B) = \frac{\frac{\text{count}(A,B)}{\cancel{|\Omega|}}}{\frac{\text{count}(B)}{\cancel{|\Omega|}}}$$

word likelihood

$$P(w_i|t_i) = \frac{\text{count}(w_i, t_i)}{\text{count}(t_i)}$$

Reminder: estimating $P(t_i | t_{i-1})$ from a corpus

Definition of conditional probability

$$P(A|B) = \frac{P(A,B)}{P(B)}$$

$$P(A|B) = \frac{\frac{\text{count}(A,B)}{\cancel{|\Omega|}}}{\frac{\text{count}(B)}{\cancel{|\Omega|}}}$$

$$P(t_i|t_{i-1}) = \frac{\text{count}(t_{i-1}, t_i)}{\text{count}(t_{i-1})}$$

POS tagging objective function

$$tt = \text{argmax}_{tt} \underset{ii}{?} \left(\frac{\text{count}(ww_{ii}, tt_{ii})}{\text{count}(tt_{ii})} \times \frac{\text{count}(tt_{ii-1}, tt_{ii})}{\text{count}(tt_{ii-1})} \right)$$

Best POS tag sequence

How often does word ww_{ii} occur with tag tt_{ii} in the corpus?

How often does tt_{ii} follow tt_{ii-1} in the corpus?

This might seem a little backwards (especially if you aren't familiar with Bayes' theorem). We're trying to find the best *tag sequence*, but we're using $PP(ww|tt)$, which seems to be predicting *words*.

This compares: "If we are expecting an **adjective** (based on the tag sequence), how likely is it that the adjective will be 'cold?'" ***versus*** "If we are expecting a **noun**, how likely is it that the noun will be 'cold?'"

DT		VBD	RB	IN	DT	NN	,
the	cold	passed	reluctantly	from	the	earth	,

"If we are expecting an **adjective**, how likely is it that the adjective will be 'cold?'" (high) **WEIGHTED BY** our chance of seeing the sequence **DT JJ** (medium)

versus

"If we are expecting a **noun**, how likely is it that the noun will be 'cold?'" (medium) **WEIGHTED BY** our chance of seeing the sequence **DT NN** (very high)

THE WINNER: NN

Multiplying probabilities

- We're multiplying a whole lot of probabilities together
- What do we know about probability values?

$$0 \le pp \le 1$$

- What happens when you multiply a lot of these together?
- This is an important consideration in computational linguistics. We need to worry about underflow.

Underflow

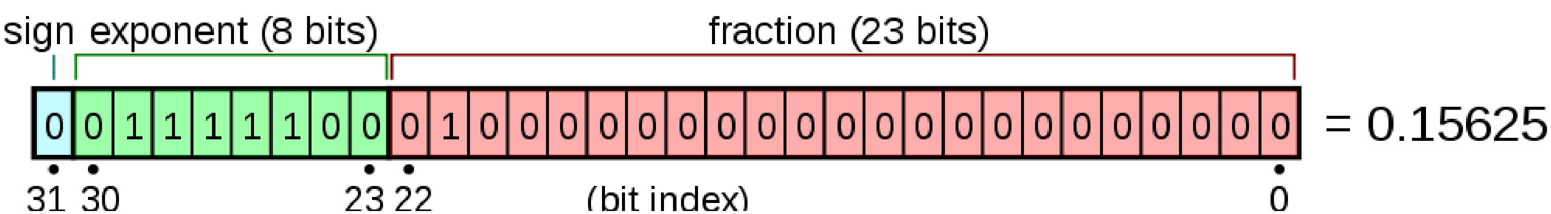
- When multiplying many probability terms together, we need to prevent underflow
  - Due to limitations in the computer's internal representation of floating point numbers, the product quickly becomes zero
- We usually work with the logarithm of the probability values
- This is known as the "log-prob"

$$= \log_{10} pp$$

IEEE 754 floating point

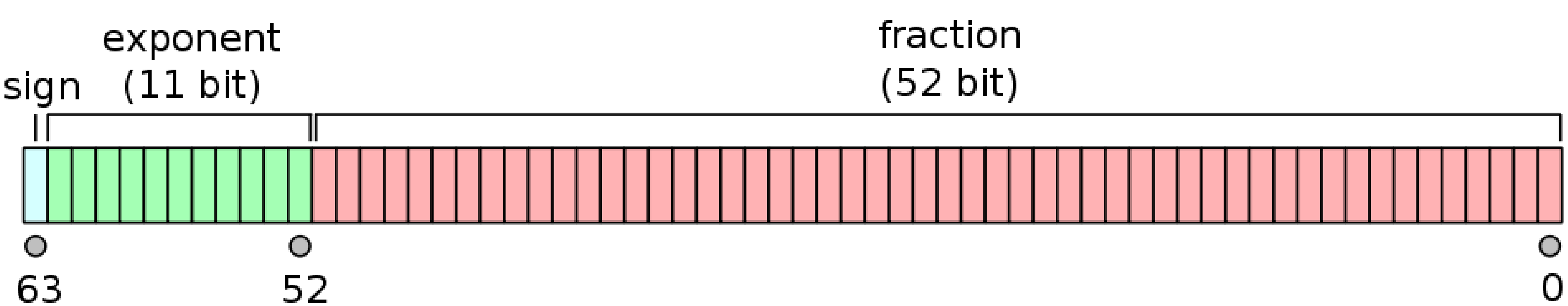
- 32-bit “single” “float”

$$1.2 \times 10^{-38}\ tttt\ 3.4 \times 10^{38}$$



- 64-bit “double”

$$\approx \pm 1.8 \times 10^{308}$$



logarithms refresher

definition: $\log_{bb} xx = yy\text{: } xx = bb^{yy}$

$$bb^{xx} \times bb^{yy} = bb^{xx+yy}$$

$$\log xxyy = \log xx + \log yy$$

$$\log \underset{ii}{?} xx_{ii} = \underset{ii}{?} \log xx_{ii}$$

$$\frac{bb^{xx}}{bb^{yy}} = bb^{xx-yy}$$

$$\log\frac{xx}{yy} = \log xx - \log yy$$

Write an expression for Bayes' theorem as log-probabilites

Bayes' theorem as log-prob

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$$\log P(A|B) = \log P(B|A) + \log P(A) - \log P(B)$$

Remember this?

$$tt = \text{argmax}_{tt} \underset{ii}{?} \left(\frac{\text{count}(ww_{ii}, tt_{ii})}{\text{count}(tt_{ii})} \times \frac{\text{count}(tt_{ii-1}, tt_{ii})}{\text{count}(tt_{ii-1})} \right)$$

$$tt = \text{argmax}_{tt} \underset{ii}{?} \left(\log \frac{\text{count}(ww_{ii}, tt_{ii})}{\text{count}(tt_{ii})} + \log \frac{\text{count}(tt_{ii-1}, tt_{ii})}{\text{count}(tt_{ii-1})} \right)$$

Wait, how can you do that, there was no "log" outside of the $\prod$!

argmax magic

- Doesn’t matter. Since argmax doesn’t care about the actual answer, but rather just the *sequence that gives it*, we can drop the overall log
  - this is valid so long as $\log x$ is a monotonically increasing function
- argmax will find the same “best” tag sequence when looking at either probabilities or log-probs because both functions will peak at the same point

$$t = \mathrm{argmax}_t \sum_i \log P(w_i \mid t_i) + \log P(t_i \mid t_{i-1})$$

Hidden Markov Model

- This is the foundation for the Hidden Markov Model (HMM) for POS tagging
- To proceed further and solve the argmax is still a challenge

$$\hat{t} = \operatorname{argmax}_{t} \sum_{i} \log P(w_i \mid t_i) + \log P(t_i \mid t_{i-1})$$

- Computing this naïvely is still $O(|T|^{n})$

Dynamic programming

- The Viterbi algorithm is typically used to decode Hidden Markov Models
  - You might get to implement it in Ling 570
- It is a dynamic programming technique
  - We maintain a trellis of partial computations
- This approach reduces the problem to $OO(|TT|^2 nn)$ time

POS Trigram model

Recall the bigram assumption:

$$PP'(tt_{ii}) \approx \prod_{ii} PP(tt_{ii} | tt_{ii-1})$$

We can improve the tagging accuracy by extending to a trigram (or larger) model

$$PP'(tt_{ii}) \approx \prod_{ii} PP(tt_{ii} | tt_{ii-2}, tt_{ii-1})$$

Data sparsity

- However, we might start having a problem if we try to get a value for $P(t_i \mid t_{i-2}, t_{i-1})$ by counting in the corpus

$$\frac{\text{count}(t_{i-2}, t_{i-1}, t_i)}{\text{count}(t_{i-2}, t_{i-1})}$$

…it was a butterfly in distress that she…

The count of this in our training set is likely to be zero

Unseens

- Our model will predict zero probability for something that we actually encounter
  – This counts as a failure of the model
- This is a pervasive problem in corpus linguistics
  – At runtime, how do you deal with observations that you never encountered during training (unseen data)?

Smoothing

- We don't want our model to have a discontinuity between something infrequent and something unseen
- Various techniques address this problem:
  - add-one smoothing
  - Good-Turing method
  - Assume unseens have probability of the rarest observation
  - Ideally, smoothing preserves the validity of your probability space

Formal grammars
Parsing

This lecture adapts some slides from:

- Andrew McCallum, (UMass)
- Chris Manning
- Jason Eisner
- Norman Landis (Fairleigh Dickinson Univ.)

Constituents

- In lecture 1, we talked about constituents
- Constituents help us organize language into structures

```
[Thing The dog] is [Place in the garden]
[Thing The dog] is [Property fierce]
[Action [Thing The dog] is chasing [Thing the cat]]
[State [Thing The dog] was sitting [Place in the garden] [Time yesterday]]
[Action [Thing We] ran [Path out into the water]]
[Action [Thing The dog] barked [Property/Manner loudly]]
[Action [Thing The dog] barked [Property/Amount nonstop for five hours]]
```

Word categories

- Traditional parts of speech

Noun	Names of things	boy, cat, truth
Verb	Action or state	become, hit
Pronoun	Used for noun	I, you, we
Adverb	Modifies V, Adj, Adv	sadly, very
Adjective	Modifies noun	happy, clever
Conjunction	Joins things	and, but, while
Preposition	Relation of N	to, from, into
Interjection	An outcry	ouch, oh, alas, psst

Substitution test

- Adjective:

  The {sad, intelligent, green, fat, …} one is in the corner.

- Noun:

  The {cat, mouse, dog} ate the bug.

- Verb:

  Kim {loves, eats, makes, buys, moves} potato chips.

Constituency

- The idea: Groups of words may behave as a single unit or phrase, called a constituent
- Sentences have parts, some of which appear to have subparts, which have subparts…
- These groupings of words that go together we will call constituents.

- e.g. Noun Phrase

  Kermit the frog

  they

  December twenty-sixth

  the reason he is running for president

Constituent Phrases

For constituents, we usually name them as phrases based on the word that heads the constituent

the man from Amherst	is a Noun Phrase (NP) because the head man is a noun
extremely clever	is an Adjective Phrase (AP) because the head clever is an adjective
down the river	is a Prepositional Phrase (PP) because the head down is a preposition
killed the rabbit	is a Verb Phrase (VP) because the head killed is a verb

Note that a word is a constituent (a little one). Sometimes words also act as phrases. In:

Joe grew potatoes.
Joe and *potatoes* are both nouns and noun phrases.

Compare with:

<u>*The man from Amherst*</u> *grew* <u>*beautiful russet potatoes*</u>*.*

We say *Joe* counts as a noun phrase because it appears in a place that a larger noun phrase could have been.

Evidence for constituency

- They appear in similar environments (before a verb)
  - *Kermit the frog comes on stage*
  - *They come to Massachusetts every summer*
  - *December twenty-sixth comes after Christmas*
  - *The reason he is running for president comes out only now.*
- But not each individual word in the constituent
  - *The comes our… *is comes out... *for comes out...
- The constituent can be placed in a number of different locations
  - On December twenty-sixth I'd like to fly to Florida.
  - I'd like to fly on December twenty-sixth to Florida.
  - I'd like to fly to Florida on December twenty-sixth.
- But not split apart:
  - *On December I'd like to fly twenty-sixth to Florida.
  - *On I'd like to fly December twenty-sixth to Florida.

Context-free grammar (CFG)

- or, "Phrase structure grammar"
- or, "Backus-Naur Form" (BNF)
- The most common way of modeling constituency
- The idea of basing a grammar on constituent structure dates back to Wilhem Wundt (1890), but not formalized until Chomsky (1956), and, independently, by Backus (1959).
- This is a particular type of formal grammar.
- Before we look at CFGs grammar, let's put them in "context…"

Formal Languages

- The set of all possible strings for an alphabet Σ is written as Σ*
- A language is a prescribed subset of Σ^*

$$LL \subset \Sigma^*$$

- Example:

  Σ = {*a*, *b*, *c*}

  Σ* = {�, *a*, *b*, *c*, *ab*, *ac*, *ba*, *bc*, *ca*, *aaa*, ...}

  one language might be the set of strings of length less than or equal to 2:

  L = {�, *a*, *b*, *c*, *aa*, *ab*, *ac*, *ba*, *bb*, *bc*, *ca*, *cb*, *cc*}

“Strings”

- Note, when we talk about formal grammars, a string can be either a string of characters or a string of words (or symbols, etc.)
- Don’t get confused by the more computer-science use/definition of the term
  - i.e. “a sequence of characters in contiguous memory, possibly zero-terminated”

Rules

- It is useful to constrain the set of strings in a language to be other than Σ^*
- Let's accept the existence of constituents
- Let's assume that constituents are defined by a set of rules, RR
- Here's how we'll constrain Σ^* : constituents can only be *juxtaposed* as permitted by RR

This introduces an ordering constraint which is prevalent in many—but not all—natural languages. But aside from LL's constraints on ordering, what practical purpose(s) might ordering serve in a language?

Modeling language

- Working towards a theoretical model of language
- So far we have:

  Σ – a set of symbols (words, letters, ...)
  R – a set of rules

- We'll need to keep track of the particular system we're defining. Such a system is called a grammar:

  $$G = \langle \Sigma, R, \ldots \rangle$$

Parsing v. Generation

- Given a configuration of rules in grammar GG, find the string(s) SS that can be formed. This is called generation

A language LL is the set of all strings that can be *generated* by grammar GG

- Given a string SS in a language LL, find a rule configuration that generates SS. This is called parsing

Formal grammar - Formal language

- A formal grammar is a (constrained) set of rules for forming strings in a language
  - The rules describe the syntax of a language: how to form strings from the symbols in Σ
  - G restricts L to some subset of Σ^*
- A formal language is the set of all strings produced by a formal grammar

$$L \subset \Sigma^*$$

- We can classify formal languages according to the *constraints* placed on the grammar rules

Grammaticality

- Grammars define formal languages
- i.e., the set of all sentences (strings of words) that can be derived by the grammar.
- Sentences in this set said to be (prescriptively) grammatical or felicitous.
- Sentences outside this set said to be (prescriptively) ungrammatical or infelicitous.

Defining grammars

- A *grammar* is defined by the tuple

$$GG = \langle VV, \Sigma, SS, RR \rangle$$

  - V is a finite set of *variables* or *preterminals*
  - Σ is a finite set of symbols, called *terminals*
  - S is in V and is called the *start symbol*
  - R is a finite set of *productions*, which are *rules* of the form

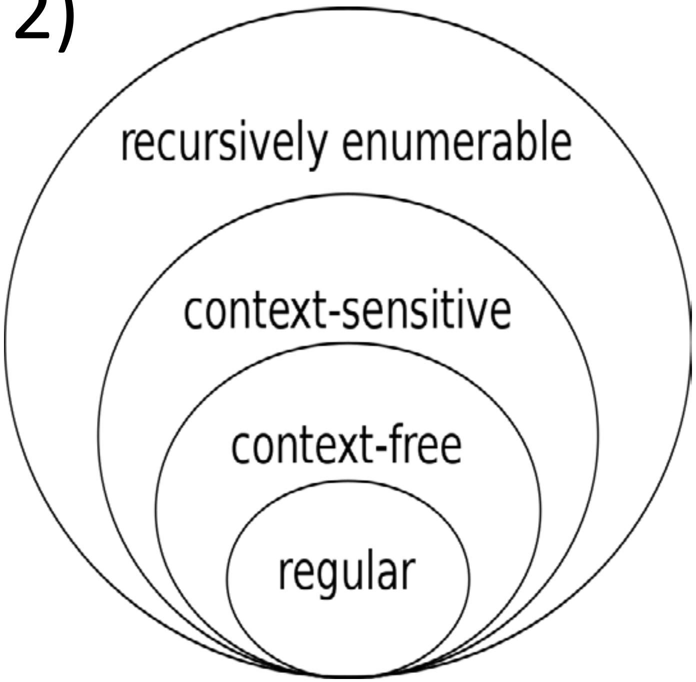
    $$\alpha\alpha \rightarrow \beta\beta$$

    where α and $\beta\beta$ are strings consisting of terminals and variables.

Types of formal grammars

- Unrestricted, recursively enumerable (type 0)
- Context-sensitive (type 1)
- Context-free grammars (type 2)
- Regular grammars (type 3)

The Chomsky hierarchy



Types of formal grammars

- Unrestricted, recursively enumerable (type 0)

  $\alpha\alpha \to \beta\beta$

  $\alpha\alpha$ and $\beta\beta$ are any string of terminals and non-terminals
- Context-sensitive (type 1)

  $\alpha\alpha XX\beta\beta \to \alpha\alpha\alpha\alpha\beta\beta$

  XX is a nonterminal; $\alpha\alpha, \beta\beta, \alpha\alpha$ are any string of terminals and non-terminals; $\alpha\alpha$ may not be empty.
- Context-free grammars (type 2)

  $XX \to \alpha\alpha$

  XX is a nonterminal; $\alpha\alpha$ are any string of terminals and non-terminals
- Regular grammars (type 3)

  $XX \to \alpha\alpha YY$

  (i.e. a *right* regular grammar)

  XX, YY are nonterminals; $\alpha\alpha$ is any string of terminals; YY may be absent.

increasing restriction

Equivalence with automata

- Formal grammar classes are defined according to the type of automaton that can accept the language
- There are various types of automatons, and many correspond to certain types of formal grammars

Grammar type	Accepted by
Recursively Enumerable	Turing machine
Context sensitive	non-deterministic linear bounded automaton (LBA)
Context Free	non-deterministic pushdown automaton (NDPA)
Regular	Finite State Automaton

Regular languages

- Let's look at the most restricted case
- A regular language (over an *alphabet* Σ) is any language for which there exists a finite state machine (finite automaton) that recognizes (accepts) it
- This class is equivalent to regular expressions (but no capture groups)

Regular Expressions and Regular Languages

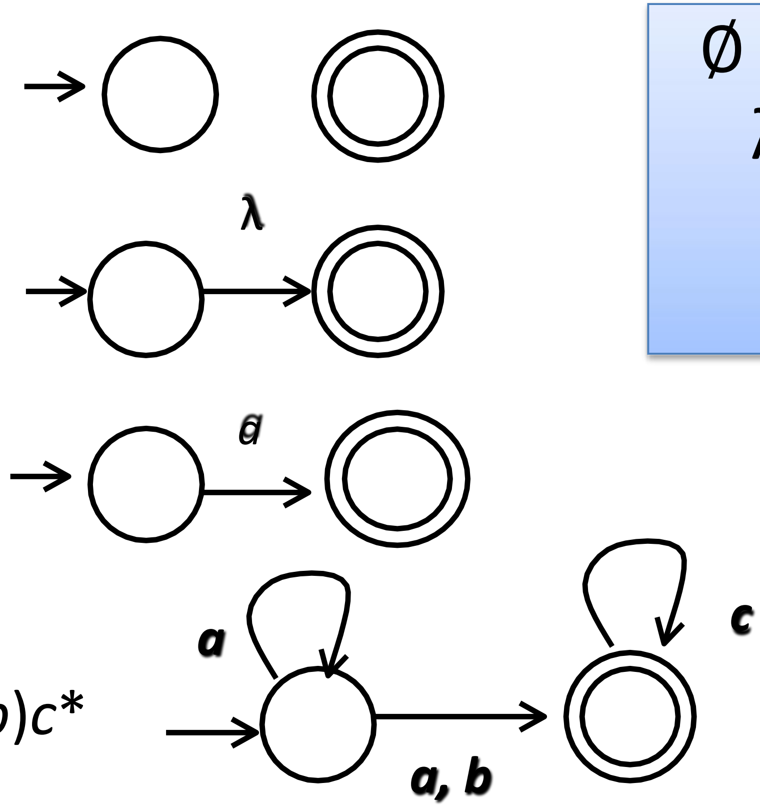
- There are simple finite automata corresponding to the simple regular expressions:

∅

λ

a

*a**(*a*+*b*)*c**



∅ - the empty set
λ – the empty string
$aa \in \Sigma$

Each of these has an initial state and one accepting state.

Regular grammars

(type 3)

- Regular grammars can be generated by FSMs
- Equivalence with RegEx
- Definition of a *right-regular grammar:*
  - Every rule in *R* is of the form
    - $A \rightarrow aB$ or
    - $A \rightarrow a$ or
    - $S \rightarrow \lambda$ (to allow λ to be in the language)
    - where *A* and *B* are variables (perhaps the same, but *B* can't be *S*) in *V*
    - and *a* is any terminal symbol

Regular grammars

(type 3)

- Example:

$$VV = \{SS, AA\}$$
$$\Sigma = \{aa, bb, cc\}$$
$$RR = \{SS \rightarrow aaSS, SS \rightarrow bbAA, AA \rightarrow \lambda\lambda, AA \rightarrow ccAA\}$$
$$SS = SS$$

RegEx: a*bc*

- Cannot express $aa^{nn}bb^{nn}$

Parsing regular grammars

(type 3)

- Parsing space: $O(1)$
- Parsing time: $O(n)$

Pumping lemma

- Two slides ago, we asserted that a regular language 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 has a value $pp \geq 1$ such that any string in the regular language LL can be decomposed into

$$xxyyxx, \qquad |yy| \geq 1, \qquad |xxyy| \leq pp, \qquad ii \geq 0$$

  such that

$$xxyy^{ii}xx \in LL,$$

- Since there's no way to satisfy this with the string $aaaaaabbbbbb$, the language $aa^{nn}bb^{nn}$ is not regular.

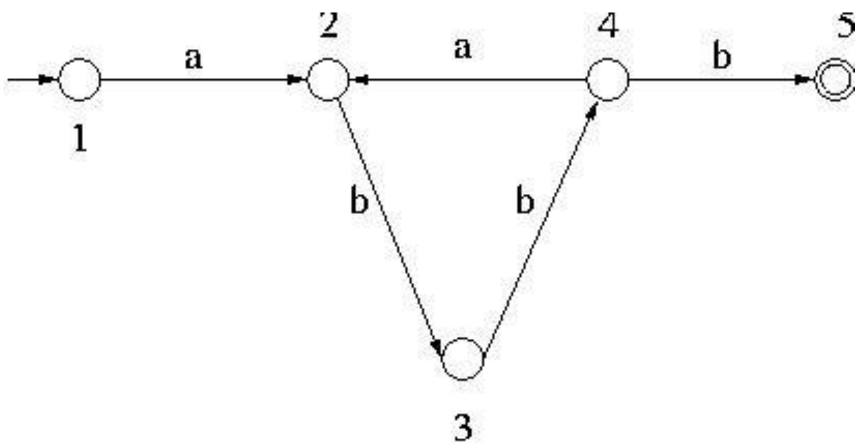
Pumping lemma

- 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 $aaaaaabbbbbb$, the language $aa^{nn}bb^{nn}$ is not regular.

Pumping lemma

- For any string in a regular language, there should be a part somewhere within the first nn 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 $aa^{nn}bb^{nn}$ is not a regular language

aaabbb

try $pp = 2$

a a abbb

a aa abbb

try $pp = 3$

aa a bbb

aa aa bbb

try $pp = 4$

aaa b bb

aaa bb bb

$\forall$ regular languages LL, $\exists\, pp$:

$$ww = xxyyxx,\ ww \in LL$$

$$|yy| \geq 1,\ |xxyy| \leq pp$$

$$\forall\, ii \geq 0 : xxyy^{ii}xx \in LL$$

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)

$$VV = \{SS\}$$
$$\Sigma = \{aa, bb\}$$
$$RR = \{SS \rightarrow aaSSbb, SS \rightarrow \varepsilon\varepsilon\}$$
$$SS = SS$$

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)

$$GG = (VV, \Sigma, RR, SS)$$

$$VV = \{ \text{ pre}-\text{terminals } vv_0, vv_1, \ldots \}$$
$$\Sigma = \{ \text{ terminals } ww_0, ww_1, \ldots \}$$
$$RR = \{ \text{ rules } rr_{ii}: vv \rightarrow \alpha\alpha \}$$
$$SS = \text{start symbol}, SS \in VV$$

$\alpha\alpha$: sequence of terminals and pre-terminals (or Ø)

LL: the language generated by GG

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

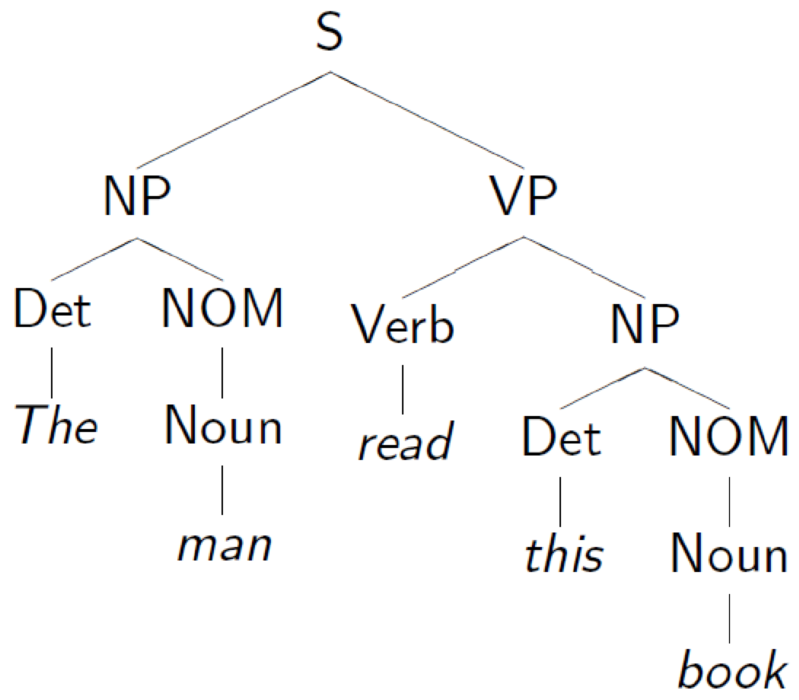
--> The man read this NOM

--> 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

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:

$$XX \rightarrow YYYY$$

$$XX \rightarrow aa$$

$$SS \rightarrow \lambda\lambda$$

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

Convert $WW \rightarrow XX\ YYaa\ YY$ to Chomsky Normal Form

CNF conversion

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

Convert the following grammar to Chomsky Normal Form:

$$SS \rightarrow AASSAA$$
$$SS \rightarrow aaBB$$
$$AA \rightarrow BB$$
$$AA \rightarrow SS$$
$$BB \rightarrow bb$$
$$BB \rightarrow \lambda\lambda$$

CNF conversion

$SS \rightarrow AASSAA$
$SS \rightarrow aaBB$
$AA \rightarrow BB$
$AA \rightarrow SS$
$BB \rightarrow bb$
$BB \rightarrow \lambda\lambda$

$SS \rightarrow AASSAA$
$SS \rightarrow UU_{aa}BB$
$AA \rightarrow BB$
$AA \rightarrow SS$
$BB \rightarrow bb$
$BB \rightarrow \lambda\lambda$
$UU_{aa} \rightarrow aa$

$SS \rightarrow AAXX$
$SS \rightarrow UU_{aa}BB$
$AA \rightarrow BB$
$AA \rightarrow SS$
$BB \rightarrow bb$
$BB \rightarrow \lambda\lambda$
$UU_{aa} \rightarrow aa$
$XX \rightarrow SSAA$

$SS \rightarrow SS'$
$SS' \rightarrow AAXX$
$SS' \rightarrow UU_{aa}BB$
$AA \rightarrow BB$
$AA \rightarrow SS'$
$BB \rightarrow bb$
$BB \rightarrow \lambda\lambda$
$UU_{aa} \rightarrow aa$
$XX \rightarrow SS'AA$

$SS \rightarrow SS'$
$SS' \rightarrow AAXX$
$SS' \rightarrow UU_{aa}BB$
$AA \rightarrow BB$
$AA \rightarrow SS'$
$BB \rightarrow bb$

$UU_{aa} \rightarrow aa$
$XX \rightarrow SS'AA$
$XX \rightarrow SS'$
$SS' \rightarrow XX$

$SS \rightarrow SS'$
$SS' \rightarrow AAXX$
$SS' \rightarrow UU_{aa}BB$
$AA \rightarrow BB$
$AA \rightarrow SS'$
$BB \rightarrow bb$
$UU_{aa} \rightarrow aa$
$XX \rightarrow SS'AA$
$XX \rightarrow SS'$
$SS' \rightarrow XX$

$SS \rightarrow XX$
$XX \rightarrow AAXX$
$XX \rightarrow UU_{aa}BB$
$AA \rightarrow BB$
$AA \rightarrow XX$
$BB \rightarrow bb$
$UU_{aa} \rightarrow aa$
$XX \rightarrow XXAA$
$\cancel{XX \rightarrow XX}$
$\cancel{SS' \rightarrow XX}$

$\cancel{SS \rightarrow XX}$
$XX \rightarrow AAXX$
$XX \rightarrow UU_{aa}BB$
$AA \rightarrow BB$
$AA \rightarrow XX$
$BB \rightarrow bb$
$UU_{aa} \rightarrow aa$
$XX \rightarrow XXAA$
$SS \rightarrow AAXX$
$SS \rightarrow UU_{aa}BB$
$SS \rightarrow XXAA$

$XX \rightarrow AAXX$
$XX \rightarrow UU_{aa}BB$
$AA \rightarrow bb$
$AA \rightarrow XX$
$BB \rightarrow bb$
$UU_{aa} \rightarrow aa$
$XX \rightarrow XXAA$
$SS \rightarrow AAXX$
$SS \rightarrow UU_{aa}BB$
$SS \rightarrow XXAA$

$XX \rightarrow AAXX$
$XX \rightarrow UU_{aa}BB$
$AA \rightarrow bb$
$\cancel{AA \rightarrow XX}$
$BB \rightarrow bb$
$UU_{aa} \rightarrow aa$
$XX \rightarrow XXAA$
$SS \rightarrow AAXX$
$SS \rightarrow UU_{aa}BB$
$SS \rightarrow XXAA$
$AA \rightarrow AAXX$
$AA \rightarrow UU_{aa}BB$
$AA \rightarrow XXAA$

all done

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

This refers to the parse tree, not the search tree. So either method can be done breadth-first or depth-first

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 GG and sentence SS if it is sound, produces every possible parse for SS, and terminates
- Often, we settle for sound but incomplete parsers
  - probabilistic parsers may be able to return the bb-best parses

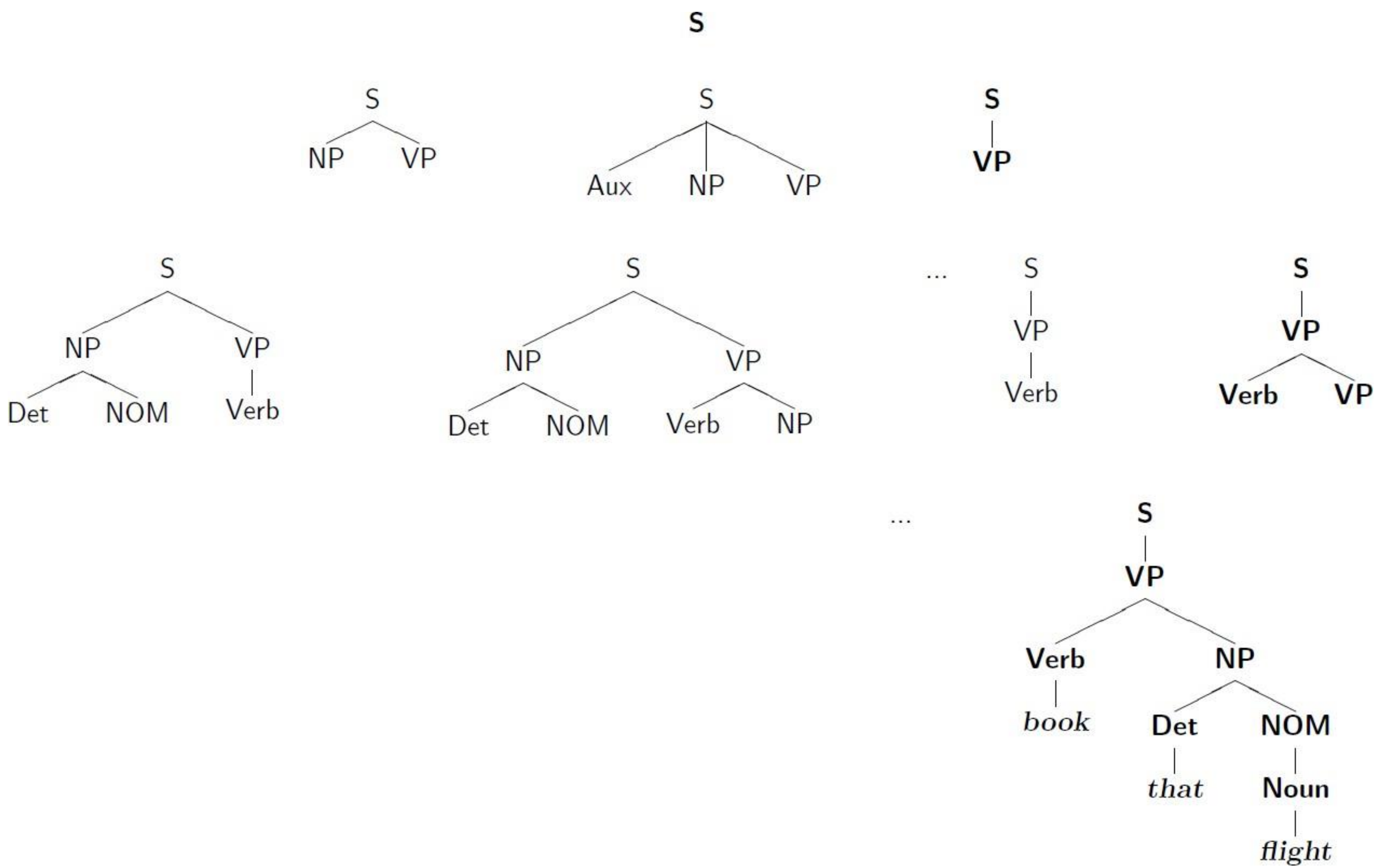
Top-down parsing

- 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

$$NNPP \rightarrow NNPP\ PPPP$$

- 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
- 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 $OO(bb^{nn})$ 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
- Get linguists on board to think about computability from the start

Next time

- Clustering
- Classifiers
- Overview of Information Theory

C# Tutorial (continued…)

IEnumerable, yield, and deferred execution

- Before describing the trie data structure, let's look at iterators which enumerate a sequence of elements

Examples in C#. If you use another language, it will be instructive to think about how to adapt the solutions to your language

- Enumeration is obvious when the data is at hand and you want to use it all:

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };

foreach (String s in data)
    Console.WriteLine(s);
```

We can pass (a reference to) the array around too, no problem

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };
// ...
ProcessSomeStrings(data);
// ...

void ProcessSomeStrings(String[] the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

What if we only want to "process" the four-letter words?

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };
// ...

List<String> filtered = new List<String>();
foreach (String s in data)
    if (s.Length == 4)
        filtered.Add(s);
ProcessSomeStrings(filtered);
// ...

void ProcessSomeStrings(List<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

This doesn't seem very nice. For one thing, we have to use more memory and waste time copying the elements we care about to a new list.

Is there a way to pass this function enough information to filter the *original list* itself, where it lies?

- Remember the non-filtered example for a second

```
void ProcessSomeStrings(String[] the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

- The processing function doesn’t really care about the fact that the data is in an array
- This violates an important programming maxim:

A flexible interface *demands the least* and *provides the most*:

- *Inputs* are as general as possible (allowing clients to supply any level of specificity, i.e. be lazy)
- *Outputs* are as specific as possible (allowing clients to capitalize on work products, i.e. be lazy).

```
void ProcessSomeStrings(String[] the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

The extra (unused) demands this function is making by asking for String[]:

- That the strings all be in memory at the same time
- That the strings be randomly accessible by an index
- That the number of strings be known and fixed before the function starts

- To modify this to comply with the maxim, we first ask:
- Q: What is the absolute minimum that this function actually needs to accomplish it’s work?
- Answer: a way to iterate strings

Interfaces

- `IEnumerable<T>` is one of many system-defined interfaces that a class can elect to implement

An interface is a named set of zero or more function signatures with no implementation(s)

- To implement an interface, a class defines a matching implementation for every function in the interface
- Interfaces are sometimes described as contracts
- You can define and use a reference to an interface just like any other object reference

Contrived Example

```
interface IPropertyGetter
{
    String GetColor();
}

class Strawberry : IPropertyGetter
{
    public String GetColor() { return "red"; }
}

class Ferrari : IPropertyGetter
{
    public String GetColor() { return "yellow"; }
}
```

- This looks like C++ class inheritance
  - yes, but it's more ad-hoc
  - C# classes can have single inheritance of other classes, and multiple inheritance of interfaces
  - Interfaces can inherit from other interfaces (not shown)

IEnumerable<T>

- This is one of the simplest interfaces defined in the BCL (base class libraries)
- This interface provides just one thing: a way to iterate over elements of type T
- All of the system arrays, collections, dictionaries, hash sets, etc. implement `IEnumerable<T>`
  – Implementing IEnumerable<T> on your own classes can be very useful, but you don't need to worry about that
  – For now, what's important is that you get to use it, because it's available on all of the system collections

IEnumerator<T>

- IEnumer**able**<T> has only one function, which allows a caller or caller(s) to obtain an *enumera__tor__* object which is able to iterate over elements
  - The actual enumerator object is an object that implements a different interface, called IEnumera**tor**<T>
  - This "factory" design allows a caller to initiate and maintain several simultaneous iterations if needed
  - The enumerator object, IEnumerator<T> can only:
    - Get the current element
    - Move to the next element
    - Tell you if you've reached the end
  - Note: There's no count
    - ICollection inherits from IEnumerable to provide this

Interfaces as function arguments

- Using interfaces as function arguments allows you to require the absolute minimum functionality the function actually needs
- In this way, the ad-hoc nature of interfaces allows us to comply with the maxim

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

Now, this function is exposing the weakest (most general) requirement possible for the processing it has to do. This provides more flexibility to callers since they can choose whatever level of specificity is convenient. The function can be used in the widest possible variety of situations.

Example

```
String[] d1 = { "able", "bodied", "cows", "don't", "eat", "fish" };
ProcessSomeStrings(d1);

List<String> d2 = new List<String> { "clifford", "the", "big", "red", "dog" };
ProcessSomeStrings(d2);

HashSet<String> d3 = new HashSet<String> { "these", "must", "be", "distinct" };
ProcessSomeStrings(d3);

Dictionary<String,int> d4 =
        new Dictionary<String, int> { { "the", 334596 }, { "in", 153024 } };
ProcessSomeStrings(d4.Keys);

void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

Python users might not be impressed, but the difference is that this is all 100% strongly typed

Iteration is efficient

- That’s cool, IEnumerable<T> lets a function not care about where a sequence of elements is coming from
  - We don’t copy the elements around
  - Iterators let us access elements right from their source
- All of those examples iterate over elements that already exist somewhere
- Is there a way to iterate over data that’s generated on-the-fly, doesn’t exist yet, or is never persisted at all?
- Yes!

Iterating over on-the-fly data

```
IEnumerable<String> GetNewsStories(int desired_count)
{
    for (int i = 0; i < desired_count; i++)
        yield return RealtimeNewswireSource.GetLatestStory();
}
```

see next slide

```
// ...
IEnumerable<String> d5 = GetNewsStories(7);
ProcessSomeStrings(d5);
// ...
```

This is exactly the same as before, but this time there's no "collection" of elements sitting anywhere

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

This function doesn't care. In fact, it can't even tell.

yield keyword

- The `yield` keyword makes it easy to define your own custom iterator functions
- Any function that contains the `yield` keyword becomes special
  - It must be declared as returning an IEnumerable<T>
  - Deferred execution means that the function's body is not necessarily invoked when you "call" it
  - It must deliver zero or more elements of type T using:

```
yield return t;
```

  - Sometime later, control may continue immediately after this statement to allow you to yield additional elements
  - It may signal the end of the sequence by using:

```
yield break;
```

Custom iterator function example

```
IEnumerable<String> GetNewsStories(int desired_count)
{
    for (int i = 0; i < desired_count; i++)
        yield return RealtimeNewswireSource.GetLatestStory();
}

// ...
IEnumerable<String> d5 = GetNewsStories(7);
ProcessSomeStrings(d5);
// ...

void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

code from this custom iterator function is *not* executed at this point.

d5 refers to an iterator that "knows how" to get a certain sequence of strings when asked

This finally demands the strings, causing our custom iterator function to execute–interleaved with this loop!

Closures

- Lambda expressions automatically capture local variables that they reference, which are then passed around as part of the lambda variable
  - Caution: languages do this differently with respect to reference (the lambda expression will modify the original value) versus value (the lambda expression has a snapshot of the value)
- This can lead to interesting scoping issues

Lambda expressions

```
// recall Select(ch => ('a' <= ch && ch <= 'z') || ch == '\'' ? ch : ' ');

String s = "Al's 20 fat-ish oxen.";
Func<Char, Char> myfunc = (ch) => ('a' <= ch && ch <= 'z') || ch == '\'' ? ch : ' ';
IEnumerable<Char> iech = s.Select(myfunc);
// iech is now a deferred enumerator for the characters in: " l's    fat ish oxen "

Func<Char, Char> myfunc = (ch) =>
    {
        if ('a' <= ch && ch <= 'z')
            return ch;
        if (ch == '\'')
            return ch;
        return ' ';
    };

Func<String, int, bool> string_is_longer_than = (s, i) => s.Length > i;

bool b = string_is_longer_than("hello", 3);     // true
```

Closure example

```
int x = 3;
Action a = () =>
    {
        Console.WriteLine(x);
    };
a();          // prints 3
x = 5;
a();          // prints 5
```

State machine example

```
using System;
using System.Collections.Generic;
using System.Linq;

static class Program
{
    static class MainClass
    {
        enum State { Zero, One, Two };

        static Dictionary<State, Func<Char, State>> machine = new Dictionary<State, Func<Char, State>>
        {
            {
                State.Zero, (ch) => { return State.Two; }
            },
            {
                State.One, (ch) => { return State.One; }
            },
        };

        static void Main(String[] args)
        {
            String s = "the string to parse";
            int i = 0;

            State state = State.Zero;
            while (i < s.Length)
                state = machine[state](s[i++]);
        }
    }
}
```

A dictionary of lambda functions

The state machine

LINQ in C#

- Sequences: IEnumerable<T>
- Deferred execution
- Type inference
- Strong typing
  - despite the 'var' keyword
  - (C# 4.0 now has the `'dynamic'` keyword, which allows true runtime typing where desired)

LINQ operators

- Filter/Quantify:
  `Where, ElementAt, First, Last, OfType`
- Aggregate:
  `Count, Any, All, Sum, Min, Max`
- Partition/Concatenate:
  `Take, Skip, Concat`
- Project/Generate:
  `Select, SelectMany, Empty, Range, Repeat`
- Set:
  `Union, Intersect, Except, Distinct`
- Sort/Ordering:
  `OrderBy, ThenBy, OrderByDescending, Reverse`
- Convert/Render:
  `Cast, ToArray, ToList, ToDictionary`

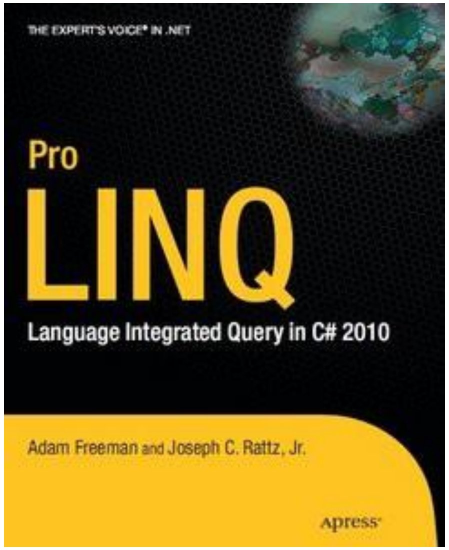
Example

```
Dictionary<String, int> pet_sym_map =
    File.ReadAllLines("/programming/analytical-grammar/erg-funcs/pet-symbol-key.txt")
    .Select(s => s.Split(sc7, StringSplitOptions.RemoveEmptyEntries))
    .Where(rgs => rgs.Length == 3)
    .Select(rgs => new { id = int.Parse(rgs[0]), sym = rgs[1].Trim().ToLower() })
    .GroupBy(a => a.sym)
    .Select(grp => grp.ArgMin(a => a.id))
    .ToDictionary(a => a.sym, a => a.id);
```

C# LINQ (Language Integrated Query)

- Declarative operations on sequences
- Recommendation:

Joseph C. Rattz, Jr. (2007) *Pro LINQ: Language Integrated Query in C# 2008*. Apress.



Programming Paradigms: Procedural

- Procedural (“imperative”) programming
  - FORTRAN (1954) grew out of hardware assembly languages, which are necessarily procedural
  - We explicitly specify the (synchronous) steps for doing something (i.e. an algorithm)

```
int Factorial(int n)
{
        int f = 1;
        for (int i = n; i > 1; i--)
                f *= i;
        return f;
}
```

Functional Programming

- A type of declarative programming

  Constraint-based syntax formalisms such as unification grammars (LFG, HPSG, …) are also declarative
- Like function definitions in math, the “program” describes asynchronous relationships
- Functions are stateless and should have no side-effects
- Immutable values
  - As a bonus, this really facilitates concurrent programming
- Scheme, Haskell, F#

F# Example

- F# interactive on patas:

```
$ mono /opt/fsharp/bin/fsi.exe --gui-
         (you must use an ANSI terminal)
```

```
> let rec factorial = function
    | 0 -> 1
    | n -> n * factorial(n -
1);; val factorial : int -> int
> factorial 5;;
val it : int =
120
> #quit;;
```