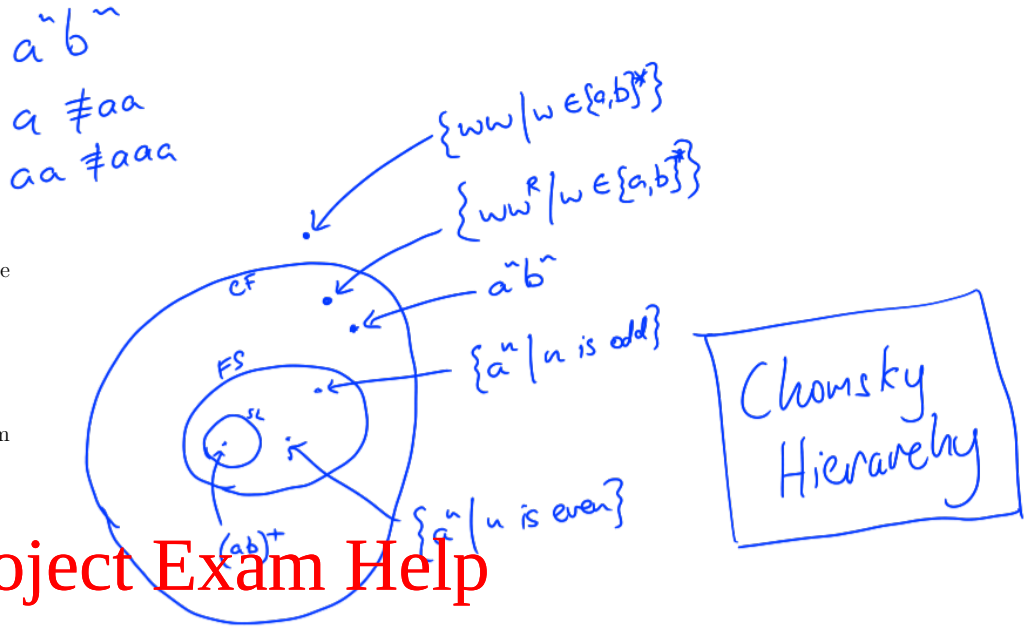
6. Context-free grammars



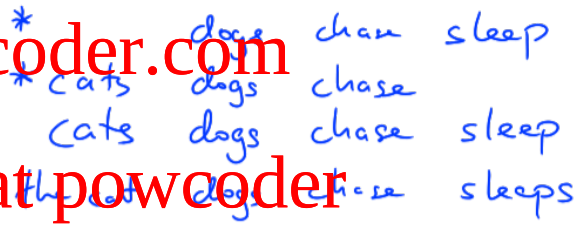
A canonical first example of a context-free grammar (CFG) is shown in (1). This grammar generates the stringset $\{a^n b^n \mid n \geq 1\}$, i.e. the stringset $\{ab, aabb, aaabbb, aaaabbbb, \dots\}$.

(1) S → aSb
S → ab

The fact that the grammar in (1) generates the string 'aaaabbbb' is due to the fact that we can start from 'S' and gradually rewrite nonterminals in accord with the rules, one step at a time, as follows:

(2) S
aSb
aaSbb
aaaSbbb
aaaabbbb

A sequence of strings related to each other like (2) is known as a *derivation*.



1 Formal definition of context-free grammars

(3) A context-free grammar (CFG) is a four-tuple (N, Σ, I, R) where:
- N is a finite set of nonterminal symbols;
- Σ, the alphabet, is a finite set of terminal symbols (disjoint from N);
- $I \subseteq N$ is the set of initial nonterminal symbols; and
- $R \subseteq (N \times (N \cup \Sigma)^*)$ is a finite set of rules; and

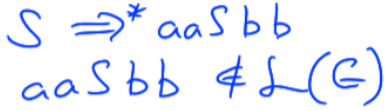
So strictly speaking, (1) is an informal representation of the following mathematical object:

(4) $$G = \big(\{\text{S}\},\quad \{\text{a},\text{b}\},\quad \{\text{S}\},\quad \{(\text{S},\text{aSb}),(\text{S},\text{ab})\}\big)$$

We often write $u \overset{G}{\Rightarrow} v$ to mean that from the string u we can derive the string v by rewriting one symbol in accord with one of the rules of the grammar G; and write $u \overset{G}{\Rightarrow}{}^* v$ to mean that from u we can derive v in one or more such steps.

So what we did above in (2) can be described in symbols like this:

(5) $\text{S} \overset{G}{\Rightarrow} \text{aSb} \overset{G}{\Rightarrow} \text{aaSbb} \overset{G}{\Rightarrow} \text{aaaSbbb} \overset{G}{\Rightarrow} \text{aaaabbbb}$
(6) $\text{S} \overset{G}{\Rightarrow}{}^* \text{aaaabbbb}$



A CFG $G = (N, \Sigma, I, R)$ generates a string $w \in \Sigma^*$ iff there's some way to derive w from some initial nonterminal symbol:

(7) $$w \in \mathcal{L}(G) \iff \bigvee_{n \in N} \Big[I(n) \wedge n \overset{G}{\Rightarrow}{}^* w \Big]$$

Two handy conventions:

- We generally use uppercase letters for nonterminal symbols and lowercase letters for terminal symbols, in which case we can recover the sets N and Σ if we're only given a collection of rules.
- Unless otherwise specified, we'll assume that there is exactly one initial nonterminal symbol, and that it is the symbol on the left hand side of first rule listed. This way the grammar can be entirely identified by listing its rules.

2 Equivalent derivations, leftmost derivations, and rightmost derivations

Now let's consider a more interesting grammar:

$N = \{NP\}$
$\Sigma = \{and, or, apples, bananas, oranges\}$

(8) NP → NP and NP
NP → NP or NP
NP → apples
NP → bananas
NP → oranges

Notice that the string 'apples and oranges' has two distinct derivations in this grammar:

(9) a. NP
NP and NP
apples and NP
apples and oranges

b. NP
NP and NP
NP and oranges
apples and oranges

And the string 'apples and oranges or bananas' has many distinct derivations, but here are two of them:

(10) a. NP
NP and NP
apples and NP
apples and NP or NP
apples and oranges or NP
apples and oranges or bananas

b. NP
NP or NP
NP and NP or NP
apples and NP or NP
apples and oranges or NP
apples and oranges or bananas

But in an important way, the two derivations in (9) seem to be just "different ways of doing the same thing", whereas the two in (10) seem "actually different". The underlying distinction is that the two derivations in (9) are both plugging together the following three facts:

(11) NP ⇒* apples and oranges
NP ⇒* apples
NP ⇒* oranges

but the two derivations in (10) plug together different collections of facts of this sort:

(12) a. NP ⇒* apples and oranges or bananas
NP ⇒* apples
NP ⇒* oranges or bananas
NP ⇒* oranges
NP ⇒* bananas

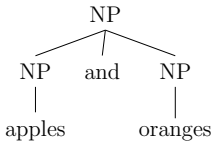
b. NP ⇒* apples and oranges or bananas
NP ⇒* apples and oranges
NP ⇒* apples
NP ⇒* oranges
NP ⇒* bananas

In the early generative linguistics literature, these individual constituency-facts like 'NP $\Rightarrow^*$ oranges or bananas' were known as "is-a relationships" (e.g. "'oranges or bananas' is a NP").[1]

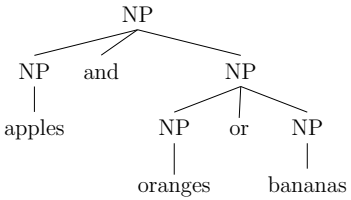
Let's call such a collection of is-a relationships an *analysis* of a string. So:

- (9a) and (9b) both correspond to the same analysis of the string 'apples and oranges', shown in (11), but
- (10a) and (10b) correspond to two distinct analyses of the string 'apples and oranges or bananas', shown in (12a) and (12b), respectively.

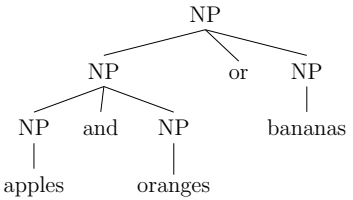
An analysis can be represented graphically by a tree.



(9) / (11)



(10a) / (12a)



(10b) / (12b)

If two derivations correspond to the same analysis, i.e. they both express the same set of is-a relations, we'll call them *equivalent derivations*.

So the relationship between derivations and analyses is many-to-one; usually we only care about distinct *analyses* of a string, and the differences between equivalent derivations (such as (9a) and (9b)) concern only the order in which the rewrites take place. We can get rid of these 'spurious choices' by adopting some fixed strategy that dictates, given any intermediate point in a derivation (such as 'NP and NP'), which nonterminal symbol must be rewritten next.

(13) a. A *leftmost derivation* is a derivation where every step rewrites the leftmost nonterminal symbol in the preceding string.
There is exactly one leftmost derivation per analysis.

b. A *rightmost derivation* is a derivation where every step rewrites the rightmost nonterminal symbol in the preceding string.
There is exactly one rightmost derivation per analysis.

The derivation in (9a) is a leftmost derivation; the derivation in (9b) is a rightmost derivation. The two derivations in (10) are both leftmost derivations.

So by considering only leftmost derivations or considering only rightmost derivations, we can leave aside the "irrelevant" differences; if we find two leftmost derivations or two rightmost derivations for a string, then it has two distinct tree structures, i.e. can be analyzed by two distinct sets of is-a relationships.

3 Finding analyses

We'll assume from here on that all CFGs are in Chomsky Normal Form (CNF), which means that each rule's right-hand side consists of *either* a single terminal symbol *or* exactly two nonterminal symbols. So in effect we'll take R to be of the form $R \subseteq N \times ((N \times N) \cup \Sigma)$. Assuming this form is computationally convenient and doesn't really restrict what we can do.[2]

$A \rightarrow B\ C$

$A \rightarrow x$

[1] See e.g. chapter 4 of Chomsky's *Syntactic Structures* (1957), or chapter 1 of Lasnik's *Syntactic Structures Revisited* (2000).

[2] Any CFG can be converted into a CFG in CNF that generates the same stringset — modulo the empty string, i.e. for any CFG G we can construct another CFG G' such that G' is in CNF and $\mathcal{L}(G') = \mathcal{L}(G) - \{\epsilon\}$.

Here's an example grammar in CNF:

(14)

VP → V NP	V → watches	$N = \{\text{VP}, \text{NP}, \text{PP}, \text{V}, \text{P}\}$
VP → VP PP	VP → watches	$\Sigma = \{\text{watches}, \text{spies}, \text{telescopes}, \text{with}\}$
NP → NP PP	VP → spies	$I = \{\text{VP}\}$
PP → P NP	NP → watches	
	NP → spies	
	NP → telescopes	
	P → with	

The important idea about how a CFG generates a string can be stated in the same form as what we saw for FSAs:

(15) $$w \in \mathcal{L}(G) \iff \bigvee_{\text{all possible trees } t} \Big[\text{string } w \text{ can generated via tree } t\Big]$$

For an FSA, the relevant notion of a path is a **linear path through the states**; here, think of a tree as a **branching path through the nonterminals**.

One way in which CFGs are more complicated than FSAs is that, even given a particular string we're interested in, we don't know what the shape(s) of the relevant path(s) might be. So spelling out what it means to consider "all possible trees t" is a bit trickier than spelling out "all possible paths p" for an FSA.

But if we leave aside the "middle" of the tree for the moment and just focus on the top and the bottom, we can at least write the following:

(16) $$x_1 x_2 \dots x_n \in \mathcal{L}(G) \iff \bigvee_{r \in N} \cdots \bigvee_{c_1 \in N} \cdots \bigvee_{c_n \in N} \Big[I(r) \wedge \cdots \wedge R(c_1, x_1) \wedge R(c_2, x_2) \wedge \cdots \wedge R(c_n, x_n)\Big]$$

For example, what we would need to evaluate to see whether the grammar in (14) will generate 'watches spies with telescopes'[3] would look something like this (still leaving aside the "middle" of the tree):

(17) $$\bigvee_{r \in N} \cdots \bigvee_{c_1 \in N} \bigvee_{c_2 \in N} \bigvee_{c_3 \in N} \Big[I(r) \wedge \cdots \wedge R(c_1, \text{watches}) \wedge R(c_2, \text{with}) \wedge R(c_3, \text{telescopes})\Big]$$

Interleaving the growing of structure with the checking of this structure against the grammar, analogously to what we did with forward and backward values for FSAs, will let us sidestep the issue of working out exactly what would need to go into this "global disjunction".

4 Inside values

For any CFG G there's a two-place predicate inside_G, relating strings to nonterminals:

(18) $\text{inside}_G(w)(n)$ is true iff the string w is derivable from the nonterminal n in grammar G

Given a way to work out $\text{inside}_G(w)(n)$ for any string and any state, we can easily use this to check for membership in $\mathcal{L}(G)$:

(19) $$w \in \mathcal{L}(G) \iff \bigvee_{n \in N} \Big[I(n) \wedge \text{inside}_G(w)(n)\Big]$$

We can arrange the values of the predicate inside_G in a table or chart. The chart in (20) shows the values of inside_G for all non-empty substrings, or *infixes*, of 'watches spies with telescopes', where G is the grammar in (14). Each cell in the chart corresponds to one of these infixes.[4]

[3] Notice that this is a length-three string belonging to Σ^*, where Σ is the set of terminal symbols in (14).

[4] So a cell in this chart, since it corresponds to an infix (and specifies a value for each nonterminal), is analogous to a *column* in the tables we saw earlier for forward and backward values.

w = watches spies with telescopes

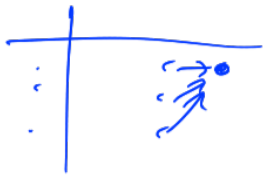
(20)

	…watches	…spies	…with	…telescopes
watches …	VP: 1 NP: 1 PP: 0 V: 1 P: 0	VP: 1 NP: 0 PP: 0 V: 0 P: 0	VP: 0 NP: 0 PP: 0 V: 0 P: 0	VP: 1 NP: 0 PP: 0 V: 0 P: 0
spies …		VP: 1 NP: 1 PP: 0 V: 0 P: 0	VP: 0 NP: 0 PP: 0 V: 0 P: 0	VP: 1 NP: 1 PP: 0 V: 0 P: 0
with …			VP: 0 NP: 0 PP: 1 V: 0 P: 1	VP: 0 NP: 0 PP: 1 V: 0 P: 0
telescopes …				VP: 0 NP: 1 PP: 0 V: 0 P: 0

inside(watches spies with telescopes)(VP) = [R(VP, V, NP) ∧ inside(watches)(V) ∧ inside(spies with telescopes)(NP)] ∨ [R(VP, VP, PP) ∧ inside(watches spies)(VP) ∧ inside(with telescopes)(PP)]

inside(spies)(VP) = 1

inside(spies with telescopes)(NP) = 1



inside(with telescopes)(NP) = 0

inside(with telescopes)(PP) = inside(with)(P) ∧ inside(telescopes)(NP) ∧ R(PP, (P, NP)) ∨ ∨ [inside(with)(C1) ∧ inside(telescopes)(C2) ∧ R(PP, C1, C2)]

We can fill this table in gradually by "working from small to big", similarly to what we saw for forward and backward values for FSAs, but with some important differences.

- Here it's the cells on the diagonal, corresponding to infixes of length one, that can be filled in just by lookups into the grammar.[5]
- We then work upwards and to the right to fill in values for longer infixes, which can be calculated by looking "just one step back" at values for shorter infixes.
- But exactly what it means to look "one step back" is a bit more complicated because of the fact that we're dealing with *all* infixes, not just prefixes and suffixes.

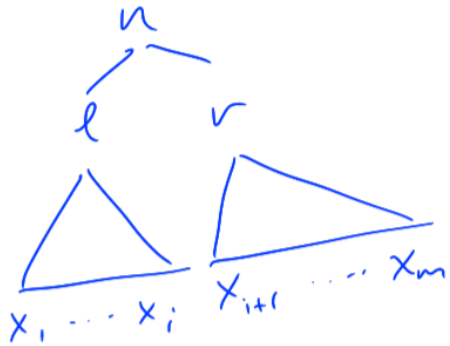
Specifically, every cell in the table can be filled in according to the following recursive definition (which is not as complicated as it looks):

n → x

(21) $$\text{inside}_G(x)(n) = R(x, n)$$

$$\text{inside}_G(x_1 \dots x_m)(n) = \bigvee_{1 \le i < m} \bigvee_{\ell \in N} \bigvee_{r \in N} \left[R(n, \ell, r) \wedge \text{inside}_G(x_1 \dots x_i)(\ell) \wedge \text{inside}_G(x_{i+1} \dots x_m)(r) \right]$$

n → l r



This algorithm for filling in a chart with inside values is known as the **CKY algorithm** (after Cocke, Kasami and Younger, who all invented/discovered it independently in the late 1960s).

[5] Like the leftmost column, representing the empty string, for forward values; and the rightmost column, representing the empty string, for backward values.

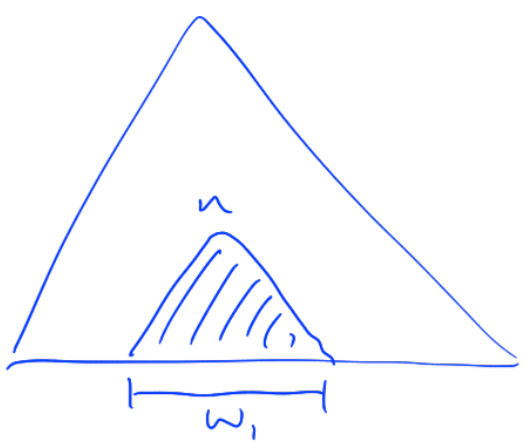
It's interesting to compare the definition in (21) with the definition we saw for backward values:

(22)
$$\text{bwd}_M(\epsilon)(q) = F(q)$$
$$\text{bwd}_M(x_1 \dots x_n)(q) = \bigvee_{q' \in Q} \left[\Delta(q, x_1, q') \wedge \text{bwd}_M(x_2 \dots x_n)(q')\right]$$

The difference is that:

- a backward value for $x_1 \dots x_n$ is broken down in terms of a single backward value, for $x_2 \dots x_n$ (plus a transition step), whereas
- an inside value for $x_1 \dots x_n$ is broken down in terms of two other inside values, for $x_1 \dots x_k$ and for $x_{k+1} \dots x_n$ (plus a transition step).

So only suffixes of a string have backward values, whereas any sub-parts of a string can have inside values.



5 Outside values

For any CFG G there's a predicate outside_G, which relates *pairs of strings* to nonterminals:

(23) $\text{outside}_G(u, v)(n)$ is true iff we can get from an initial symbol to the nonterminal n in a way that puts the string u on its left and puts the string v on its right.

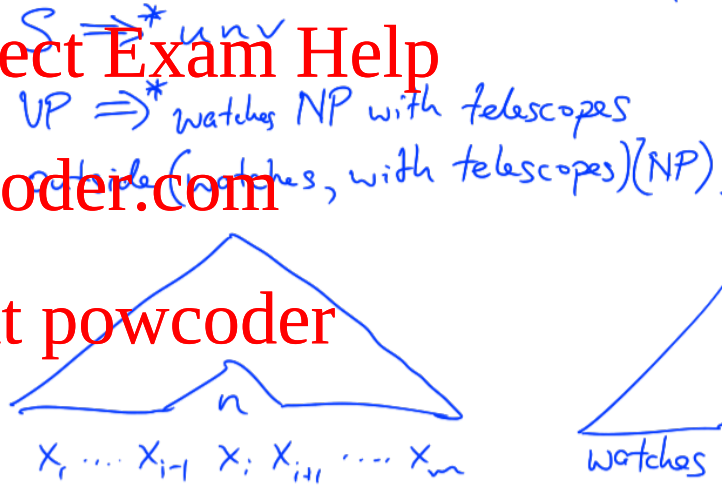
Given a way to work out $\text{outside}_G(u, v)(n)$ for any pair of strings and any nonterminal, we can easily use this to check for membership in $\mathcal{L}(G)$ — in fact we can do this in many ways, one for each symbol in the string of interest.

(24) for any $i \in \{1, \dots, m\}$:

$$x_1 \dots x_m \in \mathcal{L}(G) \iff \bigvee_{n \in N} \left[\text{outside}_G(x_1 \dots x_{i-1}, x_{i+1} \dots x_m)(n) \wedge R(n, x_i)\right]$$

Things are getting a bit more complicated here:

- An outside value concerns a *pair of strings*, because that's what you get if you take an infix — the kind of thing that has an inside value — away from a string.
  - Compared to what we saw with FSAs, this contrasts with the fact that, if you take a prefix or a suffix away from a string, you just get another string.
- Because of the asymmetry between inside and outside values, it turns out that computing outside values requires first computing inside values.
  - This means that (24) isn't itself a useful way to check well-formedness of a string; one would just use (19) instead.
  - But being able to compute outside values is (a) necessary for certain other common tasks, some of which[6] we might get to later in the course, and (b) enlightening and good for the soul.



[6] For example, the "inside-outside algorithm" for estimating the rule probabilities in a probabilistic CFG based on unparsed text. (See e.g. Manning & Schütze's *Foundations of Statistical NLP*, chapter 11.)

Here's the recursive definition of outside_G. It is not as scary as it looks — a picture helps a lot.

(25) $\text{outside}_G(\epsilon, \epsilon)(n) = I(n)$

$$\text{outside}_G(y_1 \dots y_m, z_1 \dots z_q)(n) =$$
$$\bigvee_{0<i\leq q} \bigvee_{p\in N} \bigvee_{r\in N} \Big[\text{outside}_G(y_1 \dots y_m, z_{i+1} \dots z_q)(p) \wedge R(p,n,r) \wedge \text{inside}_G(z_1 \dots z_i)(r)\Big]$$
$$\vee \bigvee_{0\leq i<m} \bigvee_{p\in N} \bigvee_{\ell\in N} \Big[\text{outside}_G(y_1 \dots y_i, z_1 \dots z_q)(p) \wedge R(p,\ell,n) \wedge \text{inside}_G(y_{i+1} \dots y_m)(\ell)\Big]$$

The key to understanding (25) is to realize that we need to deal separately with two distinct ways in which we might arrive at the nonterminal symbol n as we work downwards from an initial symbol. Either:

- n was produced as the left daughter of some other parent nonterminal symbol p, via $R(p,n,r)$ for some other right daughter r; or
- n was produced as the right daughter of some other parent nonterminal symbol p, via $R(p,\ell,n)$ for some other left daughter ℓ.

6 Inside values and outside values together

Inside values and outside values can be "snapped together" to describe the generation of a complete string, just like forward and backward values for FSAs:

(26) $$uvw \in \mathcal{L}(G) \iff \bigvee_{n\in N} \Big[\text{outside}_G(u,w)(n) \wedge \text{inside}_G(v)(n)\Big]$$

Interesting questions:

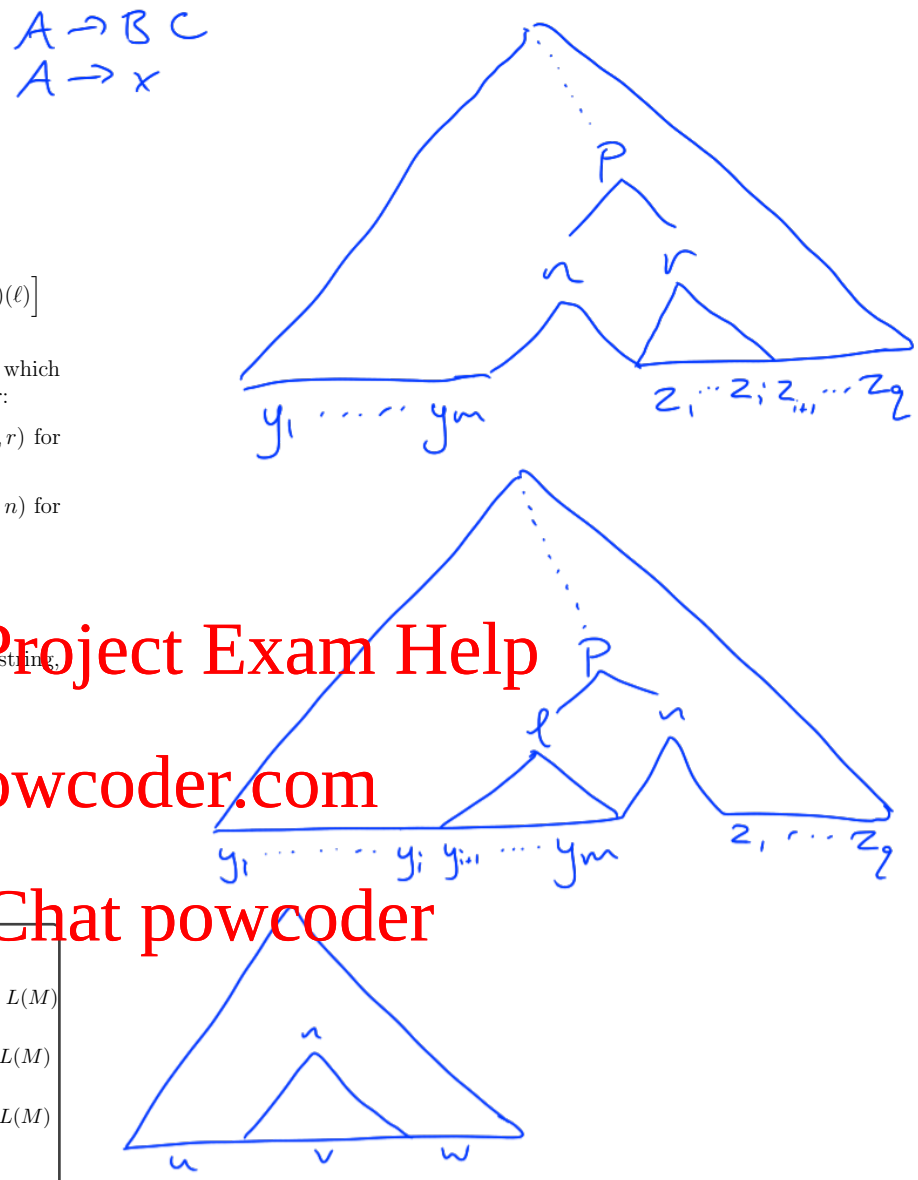
- Why does this implication only go in one direction (unlike what we saw with FSAs)?
- Why do the implications in the earlier special cases in (19) and (24) go both way?

CFGs			FSAs		
$\bigvee_{n\in N} [\text{outside}(u,w)(n) \wedge \text{inside}(v)(n)]$	$\implies$	$uvw \in L(G)$	$\bigvee_{q\in Q} [\text{fwd}(u)(q) \wedge \text{bwd}(v)(q)]$	$\iff$	$uv \in L(M)$
$\bigvee_{n\in N} [I(n) \wedge \text{inside}(w)(n)]$	$\iff$	$w \in L(G)$	$\bigvee_{q\in Q} [I(q) \wedge \text{bwd}(w)(q)]$	$\iff$	$w \in L(M)$
$\bigvee_{n\in N} [\text{outside}(u,w)(n) \wedge R(n,x)]$	$\iff$	$uxw \in L(G)$	$\bigvee_{q\in Q} [\text{fwd}(w)(q) \wedge F(q)]$	$\iff$	$w \in L(M)$

Of course, all of these calculations can be generalized to other semirings! (So there are weighted CFGs, probabilistic CFGs, etc.)

Also notice how:

- The definition of outside makes use of I and $R(c,c_1,c_2)$ rules, but not $R(c,x)$ rules. The definition of fwd makes use of I and Δ, but not F.
- The definition of inside makes use of $R(c,c_1,c_2)$ rules and $R(c,x)$ rules, but not I. The definition of bwd makes use of Δ and F, but not I.



John went to the beach yesterday

outside(John, yesterday)(VP) ∧
inside(went to the beach)(VP)

$\bigvee_n$ [outside(John went, beach yesterday)(n) ∧
inside(to the)(n)]