PART 3. CONTEXT FREE GRAMMARS AND PUSHDOWN AUTOMATA. FEATURES AND INDICES.

As we have seen, in a context free grammar, all rules are of the form $A \rightarrow \alpha$, with $A \in V_N$ and $\alpha \in V^+$ (and allowing $S \rightarrow e$ if the grammar is in reduced form),

> A context free grammar is in **Chomsky Normal Form (CNF)** iff all rules are of the form $A \rightarrow B\ C$ or $A \rightarrow a$, with $A,B,C \in V_N$, $a \in V_T$ (again, allowing $S \rightarrow e$ if the grammar is in reduced form).

Theorem: Every context free grammar is equivalent to a context free grammar in Chomsky Normal Form.

Proof:

1. Suppose we have a rule with more than one terminal symbol occurring on the right side. We choose for each terminal symbol a occurring in the rule a new non-terminal A_a, and replace in the rule a by A_a and add a new rule $A_a \rightarrow a$.

For a rule $A \rightarrow aBcdACa$ this would give:

$A \rightarrow A_aBC_cD_dACA_a$, $A_a \rightarrow a$, $C_c \rightarrow c$, $D_d \rightarrow d$.

Now we have only rules that rewrite a non-terminal into a string of non-terminals, and rules which are already in CNF.

2. Next, any rule with more than two non-terminals on the right side is replaced by a set of rules, each of which has exactly two non-terminals on the right side, a binary rule. Here we just do what we did for restricted right linear grammars:

Replace $A \rightarrow A_aBC_cD_dACA_a$ by $A \rightarrow A_aX_1$, $X_1 \rightarrow BX_2$, etc...

3. After this, the only rules left that are not in CNF are rules of the form $A \rightarrow B$.

For any such rule, we delete $A \rightarrow B$ and **add** for any rule $B \rightarrow \alpha$ a rule $A \rightarrow \alpha$.

The resulting grammar is in Chomsky Normal Form, and equivalent to the original one.

> A context free grammar is in **non-redundant form (NR)** iff every non-terminal (and every terminal) occurs at some node in an I-tree for the grammar (a tree in the set of generated trees).

Theorem: Every context free grammar is equivalent to a context free grammar in non-redundant form.

We will prove this theorem by proving a series of lemmas.

First we make a convention.

A **path** in a parse tree of G is a maximal set of nodes, linearly ordered by dominance, so, a linear path from the topnode to one of the leaves.

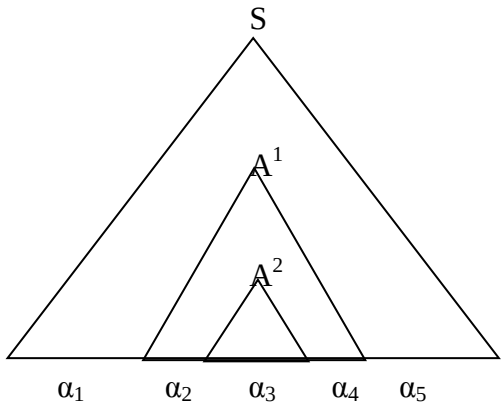
We define:

> Let p be a path in parse tree T of G
> **the length of path p**, |p| is the cardinality of the set of occurrences of **non-terminal labels** on the nodes in p.

This means that if the leaf in p is labeled by a **terminal symbol** we don't count it for the length of the path (but if the leaf is labeled by a non-terminal symbol we **do** count it). This convention makes some of our calculations a bit simpler.

Lemma 1. If a context free grammar with n non-terminals generates any string at all, it also generates **some string of terminals** with an I- tree where the length of each path is at most n.

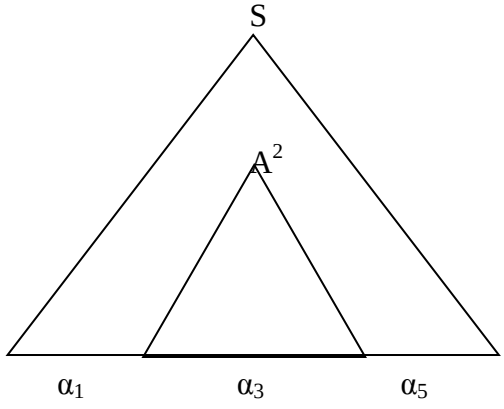
Proof:
Suppose that grammar G generates string α. Then there is an I- tree in G with topnode S and yield α. In this tree, there may be a path where a nonterminal A occurs twice, say, A^1 and A^2:



$\alpha = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$

Let us write T(X) for the subtree with topnode X.
We see:
yield(T(S)) = $\alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$
yield(T(A^1)) = $\alpha_2\alpha_3\alpha_4$
yield(T(A^2)) = α_3
S dominates A^1, A^1 dominates A^2

Since A^1 and A^2 have the same label, we can **cut out** the bit inbetween A^1 and A^2. The result is **also** a constituent structure tree generated by G:



$\alpha' = \alpha_1\alpha_3\alpha_5$

yield(T(S)) = $\alpha_1\alpha_3\alpha_5$
yield(T(A^2)) = α_3

If we do this for every non-terminal symbol that occurs twice at some path in the original T(S), we end up with a constituent structure tree of G for some $\beta \in L(G)$, in which the length of each path is at most n (n non-terminals plus one terminal on the leaf). This proves lemma 1.

Lemma 2: In a context free grammar there is an algorithm for determining whether the generated language is empty or not.

Proof:
This follows from lemma 1.
For any context free grammar with n non-terminals, there is a **finite** number k of constituent structure trees with paths not exceeding n. This means that we only need to check k trees, in order to find out whether the generated language is empty or not. This may be not very efficient, but it is an algorithm, so we have proved lemma 2.

Lemma 3: In a context free grammar there is an algorithm to determine for every non-terminal whether there is a parse tree with a terminal string as yield.

Proof:
This follows from lemma 2. You want to determine whether non-terminal A dominates any terminal string. Make A the new initial symbol of the grammar. Then the problem is equivalent to determining whether the language generated by that grammar is empty or not, and we have just shown that we have an algorithm for determining that..

Lemma 4: For every context free grammar, there is an algorithm to determine for any non-terminal symbol, whether there is a parse tree with S as topnode and that symbol in the yield.

Proof:
The proof is similar to that of lemma 1. If G has n non-terminals and there is a parse tree in G with S as topnode and non-terminal A in the yield, you can prune the tree to a parse tree for G that still has A in the yield, but doesn't have any non-terminals repeating on any path. This tree, thus has no paths longer than n. Again, there are only finitely many parse trees in G with paths no longer than n, hence, by going through those, you can just check whether S dominates A. If you find A in the yield of any one of these trees, then S dominates A, if you don't, then S doesn't dominate A in any longer tree either.

Now we can prove out theorem, which I repeat here:

Theorem: Every context free grammar is equivalent to a context free grammar in non-redundant form.

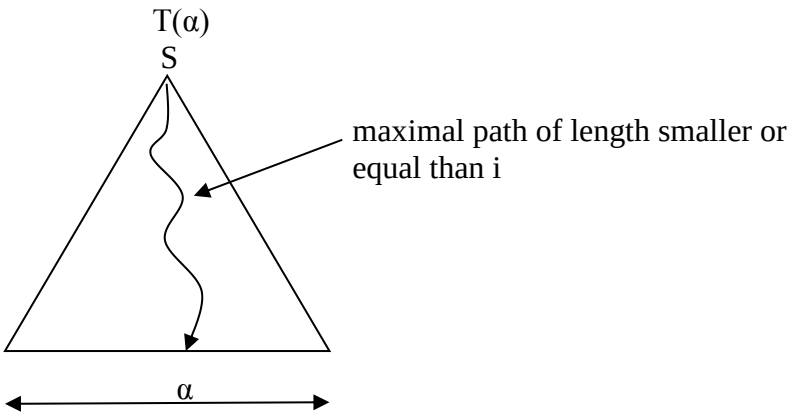
Proof:
-Check for any non-terminal whether it dominates any terminal string. If it doesn't, delete it and any rule that mentions it. That rule is useless.
-Check for any remaining non-terminal whether S dominates it. If it doesn't, delete it and any rule that mentions it. That rule is useless.
The resulting grammar is in non-redundant form and generates the same language as G (since you have only eliminated rules that weren't used in the generation of terminal strings in the first place).

Corrollary: Every context free grammar is equivalent to a context free grammar in Non-redundant Chomsky Normal Form.

Proof:
First bring the grammar in Chomsky Normal Form, then bring it in Non-redundant form.

THE STRING/PATH LEMMA.
Let G be a context free grammar in non-redundant Chomsky Normal Form.
Let $\alpha \in L(G)$ and let $T(\alpha) \in T(G)$.

If $T(\alpha)$ has no path of length greater than i, then $|\alpha| \leq 2^{i-1}$.

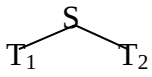


maximal length smaller or equal than 2^{i-1}

Proof: With induction.

1. i=1
If the maximal path in $T(\alpha)$ has length 1, then, since G is in non-redundant Chomsky Normal Form, α is a terminal symbol or α=e. Then $|\alpha|=1$ or $|\alpha|=0$.
Since $2^{1-1} = 2^0 = 1$, $|\alpha| \leq 2^{i-1}$.

2. i>1. Since G is in non-redundant Chomsky Normal Form, the top of T(α) is binary, and looks like:



where for some α_1,α_2, $\alpha=\alpha_1\alpha_2$ and T_1 is a parse tree with yield α_1 and T_2 is a parse tree with yield α_2

We assume: the maximal path in T(α) is at most i.

We assume as induction hypothesis that the lemma holds for trees with maximal paths **smaller than** i:
So we assume: For any parse tree with yield a terminal string and maximal path smaller or equal to **i–1**, the length of the yield is smaller or equal to $\mathbf{2^{i-2}}$.
We will prove: $|\alpha|\leq 2^{i-1}$.

The maximal path in T has length at most i. Since S is on the maximal path in T, and counts for its length, the maximal path in T_1 has length at most i–1, and the maximal path in T_2 has length at most i–1.
By the induction assumption, this means that: $|\alpha_1|\leq 2^{i-2}$ and $|\alpha_2|\leq 2^{i-2}$.
Since $\alpha = \alpha_1\alpha_2$, $|\alpha|=|\alpha_1|+|\alpha_2|$,
hence, $|\alpha|\leq 2^{i-2}+2^{i-2}$.
$2^{i-2}+2^{i-2} \leq 2^{i-1}$.
Hence, $|\alpha| \leq 2^{i-1}$.
This proves the string/path lemma.

The string/path lemma tells you that in the I- trees of a context free grammars in Chomsky normal form there is a correlation between the **height** of the tree and the **width** of the tree. More generally, it tells you that in context free grammars there is a correlation between the **length of the derivation** and the **length of the generated string**.

This is a fundamental property which helps, among others, with efficient parsing (the length of the string puts a boundary on the length of the worst case parse).

It also forms the basis for a pumping lemma for context free languages.

THE PUMPING LEMMA FOR CONTEXT FREE LANGUAGES.

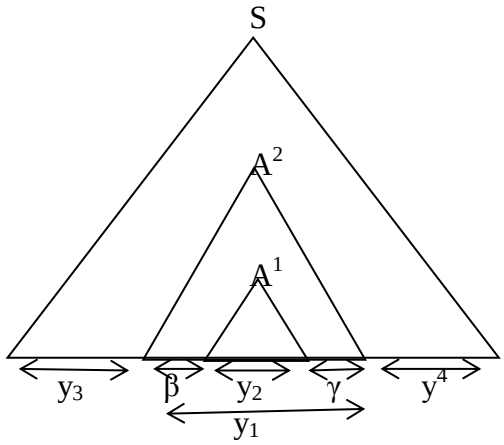
Let G be a context free grammar in non-redundant Chomsky Normal Form with k non-terminals. Let $n=2^k$.
Let $\alpha \in L(G)$ and let $|\alpha|\geq n$. This means $|\alpha|>2^{k-1}$

Using the string/path lemma, it follows that any constituent structure tree for α has at least one path of length bigger than k (i.e. at least one path with more than k non-terminals and a terminal on the leaf).

Since there are only k non-terminals in G, it follows that in any constituent structure tree for α, some non-terminal A repeats itself on the branch with length bigger than k.

Let T(α) be any such constituent structure tree, p a path of length bigger than k.
Let A be **the first label that repeats on p** if you walk up p from the leaf. Let A^1 be **the first** occurrence of A on p if you walk up p from the leaf node, and A^2 **the second** occuurrence of label A on p if you walk up p from the leaf node.

We have the following situation:

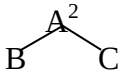


The tree dominated by A^2 is $T(A_2)$. The length of the maximal path in $T(A^2)$ is at most k+1 (A was the first repeating label on p, so the bit of p that lies in $T(A^2)$ has two occurrences of A, and further only distinct non-terminals.) Call the yield of $T(A^2)$ y_1. Then, by the string/path lemma it follows that $|y_1| \leq 2^k$. Hence, $|y_1| \leq n$.

Let us call the yield of $T(A_1)$ y_2.
By the context free format we know that $y_2 \neq e$.
Then we can write y_1 as:

$$y_1 = \beta\, y_2\, \gamma$$

Now the grammar is in non-redundant Chomsky Normal Form, this means that the top of $T(A^2)$ has the form:



By the context free format, yield(T(B)) ≠ e and yield(T(C) ≠ e.
Now A^1 is either dominated by B or by C.
-If A^1 is dominated by B, then, by the fact that yield(T(C))≠e it follows that $\gamma \neq e$.
-If A^1 is dominated by C, then, by the fact that yield(T(B))≠e it follows that $\beta \neq e$.
Hence it cannot be the case that **both** β and γ are empty.

Thus, α has $\beta y_2 \gamma$ as a substring.
Now, α may have a string y_3 to the left of $\beta y_2 \gamma$, and α may have a string y_4 to the right of $\beta y_2 \gamma$, so α itself is of the form:

$\alpha = y_3 \beta y_2 \gamma y_4$

where: 1. $|\beta\gamma|>0$
2. $|\beta y_2 \gamma| \leq n$

Now we observe that: **tree T(A^2) contains a loop**.
-Instead of of doing at node A^2 what we do in this tree, we **could have gone on directly** to the daughters of A^1. The result of doing that is a constituent structure tree of G with yield $y_3 y_2 y_4$. Hence, $y_3 y_2 y_4 \in L(G)$.
-Alternatively, we could have gone on to A^1, and instead of what we do in A^2, we could have **repeated** what we did in A^2, **and then** go to T(A^1). The result of doing that is a constituent structure tree of G with yield $y_3 \beta\beta y_2 \gamma\gamma y_4$. Hence, $y_3 \beta\beta y_2 \gamma\gamma y_4 \in L(G)$.
-In general, we could have repeated at node A^2 what we did between A^2 and A^1 as many times as we want, and then go to T(A^1).
With this we have proved:

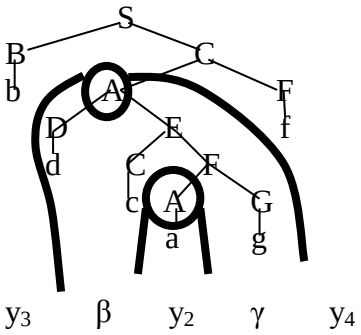
The Pumping Lemma for Context Free Languages:
Let L be a context free language. There is a number n, the **pumping constant for L**, dependent only on L (in terms of the string/path lemma) such that any string $\alpha \in L$ with $|\alpha| \geq n$ can be written as:

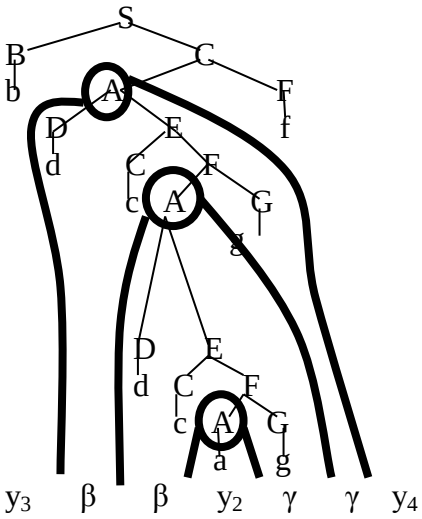
$\alpha = y_3 \beta y_2 \gamma y_4$

where: 1. $|\beta\gamma|>0$
2. $|\beta y_2 \gamma| \leq n$
3. For every $i \geq 0$: $y_3 \beta^i y_2 \gamma^i y_4 \in L$

Thus, for any context free language, we can find a constant, such that for any string **longer than** that constant, we can find a description of that string in which we can find **two substrings**, **close enough to each other, that we can pump simultaneously.**

Example: (not showing the pumping lemma, but the loop)





Applications of the pumping lemma.

Fact: $a^n b^n c^n$ is not context free.

Proof:
Assume $a^n b^n c^n$ is context free. Then it has a pumping constant z.
Choose a string α in $a^n b^n c^n$ longer than z:

a.....ab.....bc.....c

say k a's, followed by k b's, followed by k c's.

According to the pumping lemma, we should be able to write α as:
$\alpha = y_3 \beta y_2 \gamma y_3$ where β and γ are not both empty and $|\beta y_2 \gamma| \leq z$ and pumping β and γ simultaneously gives a string in $a^n b^n c^n$.

We go through all the possible cases.
Case 1. Suppose that β=e.
Then we need to find a non-empty string γ that can be pumped.
But we already saw that we couldn't find such a string for $a^n b^n$, so we certainly can't find such a string for $a^n b^n c^n$.
The case where γ=e is similar.

So we assume that both β and γ are not empty.

Case 2: β and γ consist solely of a's.
In this case, pumping β and γ will give you more a's than b's and more a's than c's, and the resulting string is not in $a^nb^nc^n$.
The cases where β and γ consist solely of b's or consist solely of c's are similar.

Case 3: β consists solely of a's and γ consists solely of b's.
In this case, pumping β and γ simultaneously will give you the same number of a's and b's, but more a's than c's, and more b's than c's. Hence the result is again not in $a^nb^nc^n$.
The cases where β consists solely of a's and γ of c's, and where β consists solely of b's and γ of c's are similar.

Case 4: One of β and γ does not solely consist of a's or solely of b's or solely of c's.
In that case, in the result of pumping, it is noy the case that all a's precede all b's and all b's precede all c's. The result is once again not in $a^nb^nc^n$.

These are all the possible cases.
We see that there **is** no division of α that satisfies the pumping lemma. Yet, if $a^nb^nc^n$ were context free there **must** be such a division, since $|\alpha|>z$.
Consequently, $a^nb^nc^n$ is not context free.

We can give similar proofs for the following facts:

Fact: $a^nb^mc^nd^m$ is not context free.

Proof: similar to that of $a^nb^nc^n$.

This is a language of **crossed dependencies.**

Note that this language must be distinguished from $a^nb^nc^md^m$. The latter is perfectly context free (since it is the product of context free languages a^nb^n and c^md^m).
It should also be distinguished from: $a^nb^mc^n$, which is also perfectly context free (Take a homomorphism that maps b onto e.)

Sometimes you cannot use the pumping lemma directly to prove that a language is not context free. For example: $a^mb^nc^nd^n$ $(m>2)$.
Take a string that is big enough, you can chose β and γ to be an a, and pumping will keep you in the language.
So crucially, the pumping lemma does **not** tell you that **if** a language satsifies the pumping lemma, it is context free, it only tells you that if a language is context free it satisfies the pumping lemma.
We can show the above language to be not context free, by using a homomorphism that maps a onto e. Then the homomorphic image is $b^nc^nd^n$ which is not context free.
Consequently, $a^mb^nc^nd^n$ $(m>2)$ is not context free.

Closure Properties of Context Free Languages.

Theorem: If A and B are context free languages, then $A \cup B$ is context free.

Proof: If $e \notin A$ and $e \notin B$: Take G_A and G_B. Make symbols disjoint,
add $S \rightarrow S_A$, $S \rightarrow S_B$. Adapt this for the cases that include e.

Theorem: If A and B are context free languages, then $A \times B$ is context free.

Proof: If $e \notin A$ and $e \notin B$: Take G_A and G_B. Make symbols disjoint, add $S \rightarrow S_A S_B$.
Adapt this for the cases that include e.

Theorem: If A is a context free language, then A* is context free.

Proof: If $e \notin A$, Change S to S_0, Add a new S and: $S \rightarrow S_0$ and $S \rightarrow S\ S_0$.
Convert to reduced form and add $S \rightarrow e$. Adapt for the case that includes e.

Theorem: The class of context free languages is **not** closed under complementation.

Proof: this follows from the next theorem.

Theorem: The class of context free languages is **not** closed under intersection.

Proof:
It is easy to show that $a^n b^n c^m$ and $a^k b^p c^p$ are context free.
Their intersection is $a^n b^n c^n$ which we will show shortly to be not context free.

Theorem: Contextfree languages are closed under substitution

Proof:
Let L be a context free language in alphabet A with grammar $G_{L.}$
Let $s: A \rightarrow pow(B^*)$ be a substution function such that for every $a \in AL$: s(a) is a context free language with grammar $G_{s(a)}$.
Let all non-terminals of all these grammars be disjoint (so $G_{s(a)}$ has start symbol $S_{s(a)}$).

We form a new grammar G':
1. Replace every rule $A \rightarrow \ldots a \ldots$ of G with $a \in A$ by: $A \rightarrow \ldots S_{s(a)} \ldots$
2. Add the rules of the $G_{s(a)}$ grammars.

The new grammar G' is obviously contextfree and generates s(L).

Corrollary: Contextfree languages are closed under homomorphisms.

Theorem: Contextfree languages are closed under inverse homomorphisms

Proof: Like the proof of the same theorem for regular languages, the proof goes by turning an automaton for L into an automaton for $h^{-1}(L)$. The construction and proof is much more complex than the earlier case, though.

Theorem: If A is a **context free** language and B is a **regular** language, then
A ∩ B is a **context free** language.

Proof: We will prove this later with an automaton.

We see that, though the class of context free languages is not closed under intersection, it **is** closed under intersection with a regular language.

Most attempted proofs to show that natural languages are not context free use this latter fact.

Not context free are **exponential languages**:

Fact: a^{2^n} is not a context free language.

This is the language:
{a, aa, aaaa, aaaaaaaa, ...}
1 2 4 8
n= 0 1 2 3

Proof: Take a string $\alpha \in a^{2^n}$ with $|\alpha| = 2^k$, where $2^k > n$
We need to find a dividion in which:
$|\beta y_2 \gamma| \leq n$
If you pump β and γ once in α you get a string α' of length $2^k + z$, where $z \leq n$.
But $2^k < 2^k + z < 2^{k+1}$,hence α' is not in a^{2^n}.

Fact: a^{n^2} is not a context free language.

This is the language:
{e, a, aaaa, aaaaaaaaa, ...}
0 1 4 9
n= 0 1 2 3
Proof: similar.

Not context free is the **string copy language**:

Fact: Let A be an alphabet.
αα = {αα: α ∈ A*}
αα is not context free.

In alphabet {a,b}, αα is the set:
{e, aa, bb, abab, baba, aaaaaa ,bbbbbb, aabaab, abaaba, baabaa, abbabb, babbab, bbabba, aaaaaaaa, bbbbbbbb,...}

This language should be distinguished from its cousins:

Let A be an alphabet:
$\boldsymbol{\alpha\alpha^R}$ = {$\alpha\alpha^R$: α ∈ A*}

This is the set of **even palindromes** on alphabet A.

In alphabet {a,b}, $\alpha\alpha^R$ is the set:
{e, aa, bb, abba, baab, aaaaaa, bbbbbb, aabbba, abaaba, baaaab, abbbba, babbab, bbaabb, aaaaaaaa, bbbbbbbb,...}

and:

Let A be an alphabet.
$\alpha\beta\alpha^R = \{\alpha\beta\alpha^R: \alpha \in A^* \text{ and } \beta \in A\}$ is the set of **odd palindromes** on alphabet A.

Though they look similar to $\alpha\alpha$, there is a crucial difference: $\alpha\alpha^R$ and $\alpha\beta\alpha^R$ are perfectly context free.
Example: Let A = {a,b,c}
A context free grammar for $\{\alpha\alpha^R: a \in \{a,b,c\}^+\}$ is:
$S\rightarrow$ aSa, $S\rightarrow$bSb, $S\rightarrow$cSc, $S\rightarrow$aa, $S\rightarrow$bb, $S\rightarrow$cc.
Convert it into reduced form and add $S\rightarrow e$, and you have a context free grammar for $\{\alpha\alpha: a \in \{a,b,c\}^*\}$.

A man, a plan, a canal, Panama.
Dennis and Edna sinned.
Able was I ere I saw Elba.
Madam, I'm Adam.

All the languages given here are context sensitive languages.

Fact: $a^nb^nc^n$ (n>0) is a context sensitive language.
A grammar generating $a^nb^nc^n$ (n>0):
$S\rightarrow abc$
$S\rightarrow abcS$
$ba\rightarrow ab$
$ca\rightarrow ac$
$cb\rightarrow bc$

This is a context sensitive grammar, since no rule is shortening.
A sample derivation shows that it generates $a^nb^nc^n$ (n>0):
S
abcS
abcabcS
abcabcabc
abcab***ac***bc
abca***ab***cbc
ab***ac***abcbc
a***ab***cabcbc
aab***ac***bcbc
aa***ab***cbcbc
aaab***bc***cbc
aaabbc***bc***c
aaabb***bc***cc = aaabbbccc No more rule applicable.

Writing a similar context sensitive grammar for $a^nb^mc^nd^m$ is simple.

Fact: $\{\alpha\alpha: \alpha \in \{a,b\}^*\}$ is context sensitive.

A context sensitive grammar for $\{\alpha\alpha: \alpha \in \{a,b\}^*\}$.

$V_T = \{a,b\}$
$V_N = \{S, X, A, B, A_0, B_0, A_e, B_e, A_m, B_m, A_{ce}, B_{ce}\}$

1. $S \to e$, $S \to aa$, $S \to bb$

2. $S \to A_0\ X$ $S \to B_0\ X$
   $X \to A\ X$ $X \to B\ X$
   $X \to A_e$ $X \to B_e$

This generates strings of the form A_0ABBBA_e
Here A_0 indicates the beginning of the string, and A_e indicates the end of the string.
Ignoring the 0,e, this bit gives you the strings in $\{\alpha \in \{A,B\}^+: |\alpha| \geq 2\}$

3. Introduce moving-A/B: A_m/B_m

$A_0 \to a\ A_m$ $B_0 \to b\ B_m$

4. Move A_m/B_m over A/B/a/b

$A_m\ A \to A\ A_m$ $B_m\ A \to A\ B_m$
$A_m\ B \to B\ A_m$ $B_m\ B \to B\ B_m$
$A_m\ a \to a\ A_m$ $B_m\ a \to a\ B_m$
$A_m\ b \to b\ A_m$ $B_m\ b \to b\ B_m$

5. Introduce copy-end:

$A_m\ A_e \to A\ A_{ce}$ $B_m\ A_e \to A\ B_{ce}$
$A_m\ B_e \to B\ A_{ce}$ $B_m\ B_e \to B\ B_{ce}$

The moving-A/B moves over A_e/B_e, turning the latter into A/B and itself becomes a copy-end A_{ce}/B_{ce}.

6. Introduce copies:

$A_m\ A_{ce} \to a\ A_{ce}$ $B_m\ A_{ce} \to a\ B_{ce}$
$A_m\ B_{ce} \to b\ A_{ce}$ $B_m\ B_{ce} \to b\ B_{ce}$

The moving-A/B moves over copy-end A_{ce}/B_{ce}, turning the latter into a/b and itself becomes the new copy-end A_{ce}/B_{ce}.

7. Introduce A_0/B_0:

$a\ A \to a\ A_0$ $a\ B \to a\ B_0$
$b\ A \to b\ A_0$ $b\ B \to b\ B_0$

8. End:

$A_{ce} \to a$ $B_{ce} \to b$

Note that the grammar is in reduced form, and no rule is shortining except for $S \to e$.
So the grammar is indeed a context sensitive grammar.

Example 1:

We first generate:	A_0ABBBA_e	
We introduce A_m:	aA_mABBBA_e	
We move A_m right:	$aABBBA_mA_e$	(this is, of course, 4 steps)
We introduce copy-end:	$aABBBAA_{ce}$	
We introduce A_0:	aA_0BBBAA_{ce}	
We introduce A_m:	aaA_mBBBAA_{ce}	
We move A_m right:	$aaBBBAA_mA_{ce}$	
We introduce a copy:	$aaBBBAaA_{ce}$	
We introduce B_0:	aaB_0BBAaA_{ce}	
We introduce B_m:	$aabB_mBBAaA_{ce}$	
We move B_m right:	$aabBBAaB_mA_{ce}$	
We introduce a copy:	$aabBBAaaB_{ce}$	
We introduce B_0:	$aabB_0BAaaB_{ce}$	
We introduce B_m:	$aabbB_mBAaaB_{ce}$	
We move B_m right:	$aabbBAaaB_mB_{ce}$	
We introduce a copy:	$aabbBAaabB_{ce}$	
We introduce B_0:	$aabbB_0AaabB_{ce}$	
We introduce B_m:	$aabbbB_mAaabB_{ce}$	
We move B_m right:	$aabbbAaabB_mB_{ce}$	
We introduce a copy:	$aabbbAaabbB_{ce}$	
We introduce A_0:	$aabbbA_0aabbB_{ce}$	
We introduce A_m:	$aabbbaA_maabbB_{ce}$	
We move A_m right:	$aabbbaaabbA_mB_{ce}$	
We introduce a copy:	$aabbbaaabbbA_{ce}$	
We end:	aabbbaaabbba	

We check that it is in αα by splitting it in the middle: aabbba aabbba.

Example 2.
The smallest strings that the algorithm in (2) derive have two non-terminals, like A_0B_e.

S
A_0X
A_0B_e
aA_mB_e
aBA_{ce}
aB_0A_{ce}
abB_mA_{ce}
$abaB_{ce}$
abab

So, the algorithm in (2) doesn't derive e, aa, bb. Instead of changing the algorithm, we just put these in by stipulation: $S\rightarrow e$, $S\rightarrow aa$, $S\rightarrow bb$.

Example 3:
If you use introduce more than one A_0/B_0, A_m/B_m simultaneously in the derivation, that makes no difference. The crucial point is that you cannot move A_m/B_m **over** another A_m/B_m, and that means that these elements will be processed correctly:

A_0AB_e
aA_mAB_e
aAA_mB_e
$aA_0A_mB_e$
$aaA_mA_mB_e$
aaA_mBA_{ce}
$aaBA_mA_{ce}$
$aaB_0A_mA_{ce}$
$aabB_mA_mA_{ce}$ You cannot move B_m over A_m.
$aabB_maA_{ce}$
$aabaB_mA_{ce}$
$aabaaB_{ce}$
aabaab

Example 4:
If you use 'end' to early, you will just get stuck:

A_0B_e
aA_mB_e
aBA_{ce}
aBa
aB_0a
abB_ma
$abaB_m$

So, indeed, we generate $\{\alpha\alpha: \alpha \in \{a,b\}^*\}$.

Once we have seen this moving and copying, it is not very difficult to show the following facts:

Fact: a^{2^n} and a^{n^2} are context sensitive languages.
Proof: This will also follow from later proofs.

DECIDABILITY THEOREM:

Let G be a context free grammar. There is an algorithm for determining of every string $\alpha \in V_T^*$ whether or not $\alpha \in L(G)$.

In fact, there are efficient decidability algorthims, we describe the most famous, the Cocke-Younger-Kashimi algorithm:

THE CYK ALGORITHM

Example from Hopcroft and Ullmann. The algorithm operates on context free grammars in non-redundant Chomsky Normal Form.

Like our example grammar:

$S \rightarrow AB$ $S \rightarrow BC$
$A \rightarrow BA$ $A \rightarrow a$
$B \rightarrow CC$ $B \rightarrow b$
$C \rightarrow AB$ $C \rightarrow a$

We want to decide whether string **baaba** is generated.

We start with reasoning (this is part of the motivation for the algorithm, not of the algorithm itself):
Since the grammar is in Chomsky Normal Form any string of more than one symbol is generates with a binary toprule of the form $S \rightarrow V\ Z$ (In this case $S \rightarrow AB$ or $S \rightarrow BC$.)
This means that the string baaba is generated as the product of two substrings, α_1 and α_2 where α_1 is generated by V and α_2 is generated by Z. The algorithm looks at all ways of binary splitting such strings.

We indicate which right sides are produced by which left sides:
$AB \Leftarrow S, C$ $BC \Leftarrow S$ $BA \Leftarrow A$ $CC \Leftarrow B$ $a \Leftarrow A, C$ $b \Leftarrow B$

Step 1: divide the string in all possible ways into **two** substrings, and divide the substrings you get again in all possible ways into **two** substrings , until you get to substrings of length 1:

length 5	**baaba**					
splits	b|aaba	ba|aba	baa|ba	baab|a		
right						
left						
length 4	**aaba**			**baab**		
splits	a|aba	aa|ba	aab|a	b|aab	ba|ab	baa|b
right						
left						
length 3	**aba**		**baa**		**aab**	
splits	a|ba	ab|a	b|aa	ba|a	a|ab	aa|b
right						
left						
length 2	**ba**		**aa**		**ab**	
splits	b|a		a|a		a|b	
right						
left						
length 1	**a**			**b**		
splits	a			b		
right						
left						

AB ⇐ S, C BC ⇐ S BA ⇐ A CC ⇐ B a ⇐ A, C b ⇐ B

Step 2.1: length 1

Determine for each string α|β what **right side** of any of the rules fits α|β, and which left side corresponds to that. We start with the bottom row:

length 5	**baaba**					
splits	b|aaba	ba|aba	baa|ba	baab|a		
right						
left						
length 4	**aaba**			**baab**		
splits	a|aba	aa|ba	aab|a	b|aab	ba|ab	baa|b
right						
left						
length 3	**aba**		**baa**		**aab**	
splits	a|ba	ab|a	b|aa	ba|a	a|ab	aa|b
right						
left						
length 2	**ba**		**aa**		**ab**	
splits	b|a		a|a		a|b	
right						
left						
length 1	**a**			**b**		
splits	a			b		
right	a			b		
left	**A, C**			**B**		

AB ⇐ S, C BC ⇐ S BA ⇐ A CC ⇐ B a ⇐ A, C b ⇐ B

Step 2.2: length 2

length 5	**baaba**					
splits	b|aaba	ba|aba	baa|ba	baab|a		
right						
left						
length 4	**aaba**			**baab**		
splits	a|aba	aa|ba	aab|a	b|aab	ba|ab	baa|b
right						
left						
length 3	**aba**		**baa**		**aab**	
splits	a|ba	ab|a	b|aa	ba|a	a|ab	aa|b
right						
left						
length 2	**ba**		**aa**		**ab**	
splits	b|a		a|a		a|b	
right	BA, BC		AA, AC, CA, CC		AB, CB	
left	**S, A**		**B**		**S,C**	
length 1	**a**			**b**		
splits	a			b		
right	a			b		
left	**A, C**			**B**		

AB ⇐ S, C BC ⇐ S BA ⇐ A CC ⇐ B a ⇐ A, C b ⇐ B

Step 2.3: length 3

length 5	**baaba**					
splits	b|aaba	ba|aba	baa|ba	baab|a		
right						
left						
length 4	**aaba**			**baab**		
splits	a|aba	aa|ba	aab|a	b|aab	ba|ab	baa|b
right						
left						
length 3	**aba**		**baa**		**aab**	
splits	a|ba	ab|a	b|aa	ba|a	a|ab	aa|b
right	AS, AA, CS, CA	SA, SC CA, CC	BB	SA, SC AA, AC	AS, AC CS, CC	BB
left	**Ø**	**B**	**Ø**	**Ø**	**B**	**Ø**
length 2	**ba**		**aa**		**ab**	
splits	b|a		a|a		a|b	
right	BA, BC		AA, AC, CA, CC		AB, CB	
left	**S, A**		**B**		**S, C**	
length 1	**a**			**b**		
splits	a			b		
right	a			b		
left	**A, C**			**B**		

AB ⇐ S, C BC ⇐ S BA ⇐ A CC ⇐ B a ⇐ A, C b ⇐ B

Step 2.4 length 4

length 5	**baaba**					
splits	b|aaba	ba|aba	baa|ba	baab|a		
right						
left						
length 4	**aaba**			**baab**		
splits	a|aba	aa|ba	aab|a	b|aab	ba|ab	baa|b
right	AB, CB	BS, BA	BA, BC	BB	SS, SC AS, AC	Ø
left	**S, C**	**A**	**S, A**	**Ø**	**Ø**	**Ø**
length 3	**aba**		**baa**		**aab**	
splits	a|ba	ab|a	b|aa	ba|a	a|ab	aa|b
right	AS, AA, CS, CA	SA, SC CA, CC	BB	SA, SC AA, AC	AS, AC CS, CC	BB
left	**Ø**	**B**	**Ø**	**Ø**	**B**	**Ø**
length 2	**ba**		**aa**		**ab**	
splits	b|a		a|a		a|b	
right	BA, BC		AA, AC, CA, CC		AB, CB	
left	**S, A**		**B**		**S, C**	
length 1	**a**			**b**		
splits	a			b		
right	a			b		
left	**A, C**			**B**		

AB ⇐ S, C BC ⇐ S BA ⇐ A CC ⇐ B a ⇐ A, C b ⇐ B

Step 2.5 length 5

length 5	**baaba**					
splits	b|aaba	ba|aba	baa|ba	baab|a		
right	BS, BC	SB, AB	Ø	Ø		
left	**S**	**S, C**	**Ø**	**Ø**		
length 4	**aaba**			**baab**		
splits	a|aba	aa|ba	aab|a	b|aab	ba|ab	baa|b
right	AB, CB	BS, BA	BA, BC	BB	SS, SC AS, AC	Ø
left	**S, C**	**A**	**S, A**	**Ø**	**Ø**	**Ø**
length 3	**aba**		**baa**		**aab**	
splits	a|ba	ab|a	b|aa	ba|a	a|ab	aa|b
right	AS, AA, CS, CA	SA, SC CA, CC	BB	SA, SC AA, AC	AS, AC CS, CC	BB
left	**Ø**	**B**	**Ø**	**Ø**	**B**	**Ø**
length 2	**ba**		**aa**		**ab**	
splits	b|a		a|a		a|b	
right	BA, BC		AA, AC, CA, CC		AB, CB	
left	**S, A**		**B**		**S, C**	
length 1	**a**			**b**		
splits	a			b		
right	a			b		
left	**A, C**			**B**		

AB ⇐ S, C BC ⇐ S BA ⇐ A CC ⇐ B a ⇐ A, C b ⇐ B

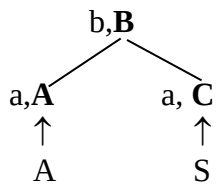
Step 3: baaba is generated by the grammar if any *left*-box on row *length 5* contains **S**.

We see that **baaba** is generated by the grammar.

Let us count the number of things we need to do in order to go through the three steps of the algorithm completely.

Step 1: Make binary divisions: 21 steps
Step 2: Determine right sides and left sides: 58 steps

Arguably, determining the left and right sides for **b|a** counts as three steps:
namely, determine the three boldface nodes in the folllowing tree:



That is, we don't need to think of the computation of the left side as a different step, because we can let the algorithm write that down as part of writing down the paths BA and BC.

Step 3: Check that **S** is in *left* at stage 5: 1 step (because we look left to right)
Total: 80 steps

We may quibble about the exact definition of a steps, and the number may vary a bit depending on whether you look left to right or right to left (in **step 3**), but clearly, the number of steps **k** is: $5^2 < \mathbf{k} < 5^3$.

This is a general property of the CYK algoritm, it runs (at worst) in cubic time. Faster algorithm can be given which run in a bit more than quadratic time. Faster than that would only be possible if you were able to speed up the general algorithm for forming partitions, which – if you were to do it, would make you rich and famous.

PUSHDOWN STORAGE AUTOMATA

A **pushdown storage automaton** consists of a **finite state automaton** extended with a **limited memory** wich takes the form of a **pushdown storage tape**.
Symbols can be written on the storage tape at a stage of the derivation, and retrieved at another stage of the derivation. We think of the store vertically. The automaton has a reading head for the storage tape which always reads the topmost symbol of the store. It has a bottom, so the automaton starts out reading the bottom of the store. When a symbol is stored, it is put on top of the store, and, since the automaton always reads the top of the store, the store is pushed down with each symbol stored. Again, since the automaton can only ever read the top of the store, symbols are removed from the store from the top. This means that the store works on the principle:

First in, last out.

Apart from the fact that the store has a bottom, it has unlimited storage capacity: you can store as much on it as you want.

There are several different, but equivalent formulations of pushdown storage automata in the literature. The formulation we give is meant to be easy to use.
It differs from our earlier formulation of finite state automata in that we allow the empty string to occur in transitions. We will give such occurrences a special interpretation, and use them to write one rule instead of many. But pay attention to the instructions on using e!

A **pushdown storage automaton** is a tuple $M = \langle V_I, S, S_0, F, \delta, V_O, Z_0, \sigma \rangle$ where:

1. V_I, the **input alphabet**, is a finite alphabet.
   The alphabet that strings on the input tape are written in.
2. S is a finite set of **states**.
3. $S_0 \in S$, the **initial state**.
4. $F \subseteq S$, the set of **final states**.

5. δ is a finite set of transitions, specified below.
6. V_O, the **storage alphabet**, is a finite alphabet.
   The alphabet that strings on the storage tape are written in.
7. $Z_0 \in V_O$, Z_0 is a symbol indicating the **bottom of the store**.
8. $\sigma \notin V_I \cup V_O$, σ is the **erase symbol**.

We specify δ:

$$\delta \subseteq (V_I \cup \{e\}) \times S \times (V_O \cup \{e\}) \times S \times (V_O \cup \{e,\sigma\})$$

This means that δ is a **finite set of transitions** of the form:
$(\alpha,S_i,\beta)\rightarrow(S_k,\gamma)$
where: 1. $S_i,S_k \in S$.
2. $\alpha \in V_I \cup \{e\}$.
3. $\beta \in V_O \cup \{e\}$.
4. $\gamma \in V_O \cup \{e,\sigma\}$.

As before, we specify the invariable parts of the automaton:

1. Every automaton has an input tape, on which a string in the input alphabet is written.
2. Every automaton has a storage tape, on which initially only Z_0 is written.
3. Every automaton has a reading head for the input tape which reads one symbol at a time.
4. Every automaton has a reading head for the storage tape which always reads the topmost symbol on the storage tape.
5. Every computation **starts** while the automaton is **in the initial state S_0**, **reading the first symbol** of the input string on the input tape **and reading Z_0** on the storage tape.
6. We assume that **after** having read the last symbol of the input string, the automaton reads e.
7. At each computation step the automaton follows a transition.
   $(\alpha,S_i,\beta)\rightarrow(S_k,\gamma)$

With this transition, the automaton can perform the following computation step:

Computation step:
If the automaton is in state S_i and reads α on the input tape,
and reads β on the storage tape, it swiches to state S_k and performs the following instruction:

1. If $\alpha \in V_I$ and $\beta \in V_O$ and $\gamma \in V_O$, then:
-the reading head on the input tape moves to the next symbol of the input.
-γ is put on the top of the store.
-the reading head on the store reads γ.

2. If $\alpha \in V_I$ and $\beta \in V_O$ and $\gamma = \sigma$, then:
-the reading head on the input tape moves to the next symbol of the input.
-β is removed from the top of the store.
-The reading head on the store reads the symbol that was below β on the store.
We take this to mean that if β was Z_0, the reading head on the store reads nothing (not even e) and any further transition (α,F,β) is undefined.

3. If α=e we carry out the instruction exactly as under (1) and (2), REGARDLESS OF WHAT THE READING HEAD FOR THE INPUT TAPE READS ON THE INPUT TAPE, WITH THE EXCEPTION THAT THE READING HEAD FOR THE INPUT TAPE DOES NOT MOVE TO THE NEXT SYMBOL ON THE INPUT TAPE.

4. If β=e we carry out the instruction exactly as under (1) and (2), REGARDLESS OF WHAT THE READING HEAD FOR THE STORAGE TAPE READS ON THE TOP OF THE STORAGE TAPE. Thus, if $\gamma \in V_O$, we add γ to the top of the store, regardless of what there was before, and if $\gamma=\sigma$, we erase from the store whatever symbol was on top of the store.

5. If γ=e we carry out the instruction exactly as under (1) and (2), except that we do not change the top of the store.

6. We interpret the constraints in (4), (5) and (6) cumulatively.
This means, for example, that for a transition $(e,S_i,e)\rightarrow(S_k,\sigma)$, in a state S_i, reading a on the input tape and Z_0ab on the storage tape, we switch to S_k, leave the input reading head on a, and erase b from the storage tape. So the new storage tape is Z_0a, and the reading head on the storage tape reads a.

The important thing to note is that the reading head on the input tape **only doesn't move to the next symbol if α=e.**

As before,

The automaton **halts** iff there is no transition rule to continue.

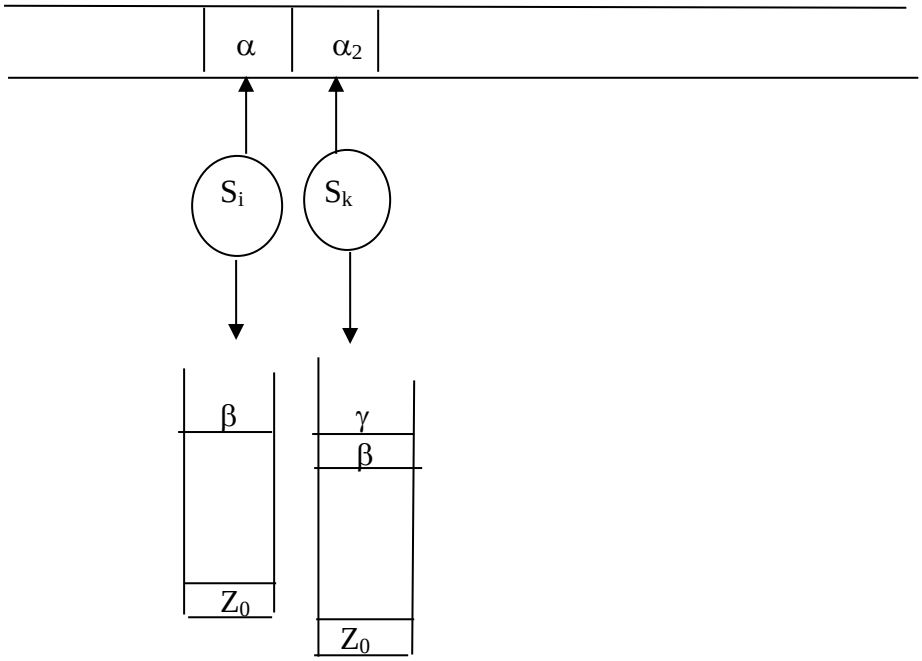
Let $\alpha \in V_I^*$.
A **computation path for α in** M is a sequence of computation steps beginning in S_0 reading the first symbol of α on the input tape and reading Z_0 on the storage tape, following instructions in δ until M halts.

A **computating path processing α in M** is a computation path for α in M which halts with the reading head on the input tape reading e **after** it has read all symbols in α.

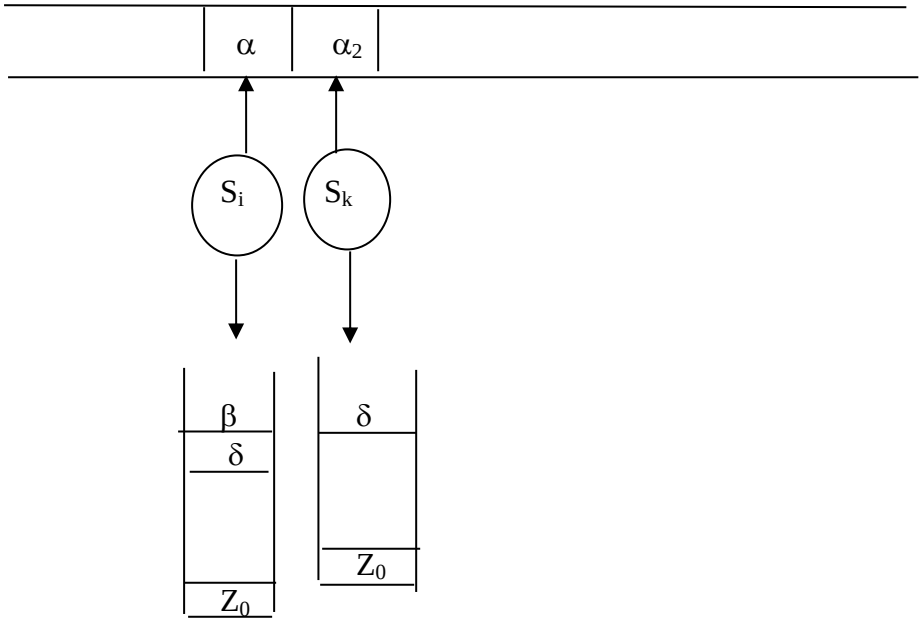
$\alpha \in V_I^*$ is **accepted by** M iff there is a computation path processing α in M where at the end of the path:
1. **M is in a final state**.
2. **The store is empty.** This means, Z_0 has been removed from the store.

In sum:

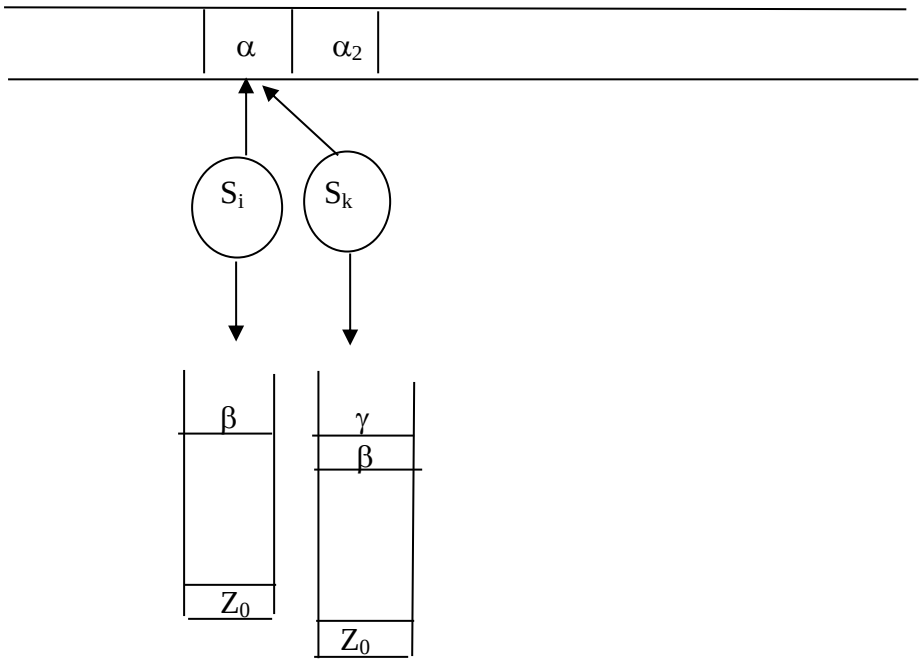
1. $(\ \alpha,\ \ S_i,\ \ \beta) \rightarrow (\ S_k,\ \gamma\ )$
   $V_I \qquad V_O \qquad\quad V_O$



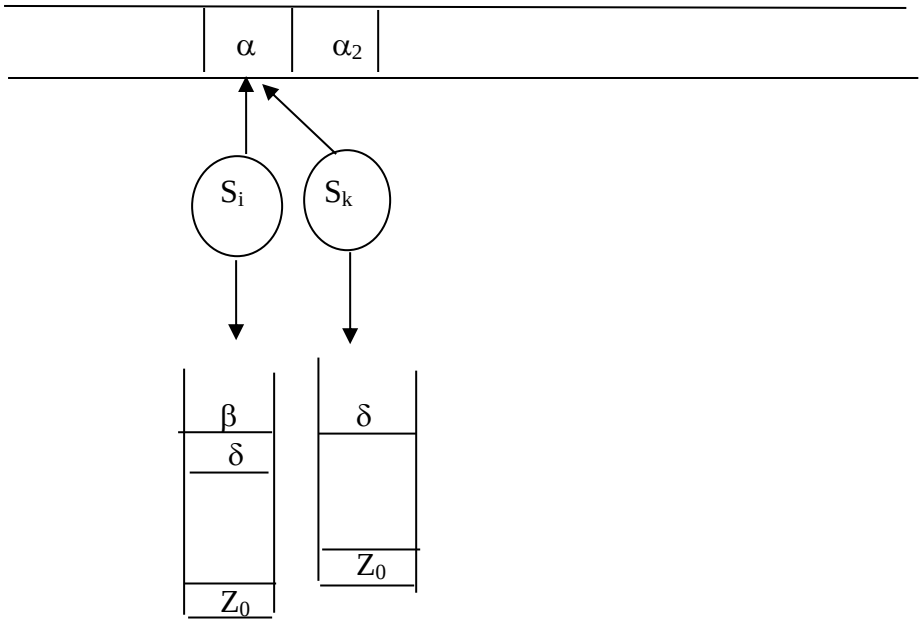
2. $(\ \alpha,\ \ S_i,\ \ \beta) \rightarrow (\ S_k,\ \sigma\ )$
   $V_I \qquad V_O$



3. (e, S_i, β) $\rightarrow$ (S_k, γ)



4. (e, S_i, β) $\rightarrow$ (S_k, σ)



5. (α, S_i, e) $\rightarrow$ (S_k, γ)
The same as under 1-4, but do γ independent of what is on top of the store.

6. (α, S_i, e) $\rightarrow$ (S_k, e)
The same as unde r1-5 but don't change the store.

Important: in (α, S_i, β) $\rightarrow$ (S_k, γ) you don't move on the input tape iff α = e.

Example: a^nb^n $(n>0)$
$V_I = \{a,b\}$
$V_O = \{Z_0,1\}$
$S = \{S_0,S_1,S_2\}$
$F = \{S_2\}$
$\delta = \{\delta_1,\delta_2,\delta_3,\delta_4\}$

δ_1: $(a,S_0,e)\rightarrow(S_0,1)$
δ_2: $(b,S_0,1)\rightarrow(S_1,\sigma)$
δ_3: $(b,S_1,1)\rightarrow(S_1,\sigma)$
δ_4: $(e,S_1,Z_0)\rightarrow(S_2,\sigma)$

We compute: aaabbb

Step 1. a a a b b b e
↑
S_0
↓
Z_0

Step 2. a a a b b b e
↑
S_0
↓
1
Z_0

Step 3. a a a b b b e
↑
S_0
↓
1
1
Z_0

Step 4. a a a b b b e
↑
S_0
↓
1
1
1
Z_0

Step 5. a a a b b b e
↑
S_1
↓
1
1
Z_0

Step 6. a a a b b b e
↑
S_1
↓
1
Z_0

Step 7. a a a b b b e
↑
S_1
↓
Z_0

Step 8. a a a b b b e
↑
S_2
↓

M halts in a final state while reading e on the input and on the store, S_2 is a final state, hence M accepps aaabbb.

We compute: aaabbbb

We get, as before, to:

Step 6: a a a b b b b e
↑
S_1
↓
1
Z_0

Step 7: a a a b b b b e
↑
S_1
↓
Z_0

Step 8: a a a b b b b e
↑
S_2
↓

We applied δ_4. Note that the reading head on the input did not move on.
M halts in a final state with an empty store, but it doesn't accept the string, because the path is not a computation path **processing** the string (M is not reading e, **after** all the symbols have been read, since it got stuck while reading the last symbol.)
So aaabbbb is not accepted.

We compute: aaabb

As before, we get to:

Step 5. a a a b b e
↑
S_1
↓
1
1
Z_0

Step 6. a a a b b e
↑
S_1
↓
1
Z_0
This time M halts after having read the whole input, but it is not in a final state and the store is not empty. So M rejects aaabb.

Example: $\alpha c \alpha^R$ with $\alpha \in \{a,b\}^*$.
$V_I = \{a,b,c\}$
$V_O = \{Z_0, a,b,c\}$
$S = \{S_0, S_1, S_2\}$
$F = \{S_2\}$
$\delta = \{\delta_1,\delta_2,\delta_3,\delta_4,\delta_5,\delta_6\}$

δ_1: $(a,S_0,e)\rightarrow(S_0,a)$
δ_2: $(b,S_0,e)\rightarrow(S_0,b)$
δ_3: $(c,S_0,e)\rightarrow(S_1,e)$
δ_4: $(a,S_1,a)\rightarrow(S_1,\sigma)$
δ_5: $(b,S_1,b)\rightarrow(S_1,\sigma)$
δ_6: $(e,S_1,Z_0)\rightarrow(S_2,\sigma)$

We compute: babaabacabaabab

For ease we write the store horizontally.

Step 1: b a b a a b a c a b a a b a b e

↑

S_0

↓

Z_0

Step 2: b a b a a b a c a b a a b a b e

↑

S_0

↓

Z_0b

Step 3: b a b a a b a c a b a a b a b e

↑

S_0

↓

Z_0ba

Step 4: b a b a a b a c a b a a b a b e

↑

S_0

↓

Z_0bab

Step 8: b a b a a b a c a b a a b a b e

↑

S_0

↓

Z_0babaaba

Step 9: b a b a a b a c a b a a b a b e

↑

S_1

↓

Z_0babaaba

Step 10: b a b a a b a c a b a a b a b e

↑

S_1

↓

Z_0babaab

Step 16: b a b a a b a c a b a a b a b e

↑

S_1

↓

Z_0

Step 17: b a b a a b a c a b a a b a b e
↑
S_2
↓

M accepts.
So, the intuition is: M stores the first part of the string while reading. This will put an inverse copy of the first part in the store. At c, M switches from reading and storing to matching, symbol by symbol the inverse copy in the input with the inverse copy in the store.
Since the automaton always reads the top symbol of the store, this algorithm wouldn't work for αcα: when c switches to matching, the **end** of the read α is on top of the store, not the beginning. Since you don't have access to what is deeper down in the store, you cannot match.
(You could, if you change the automaton to an automaton that always reads the bottom of the store. But such automata wouldn't accept $\alpha c \alpha^R$.)

> Let M be a pushdown storage automaton with transition relation δ.
> $\boldsymbol{\delta^{CL}}$ is the closure of δ under entailed transitions.
> Obviously you generate the same language with δ as with δ^{CL}, since δ^{CL} only makes the conventions explicit.

This means that if δ contains, say, a transition $(e,S_i,a)\rightarrow(S_j,\sigma)$ and the input alphabet is {a,b}, then the transitions $(a,S_i,a)\rightarrow(S_j,\sigma)$ and $(b,S_i,a)\rightarrow(Sj,\sigma)$ are in δ^{CL}.

> A **deterministic** pushdown storage automaton is a pushdown storage automaton M where δ^{CL} is a partial function.

As before, we identify **non-deterministic** pushdown storage automata with pushdown storage automata.
We call the languages accepted by pushdown storage automata **pushdown storage languages.** And we use the terms (**non-deterministic) pushdown storage languages** and **pushdown storage languages**.

Fact: There are pushdown storage languages that are not deterministic pushdown storage languages.

Example: $\alpha\alpha^R$ is a pushdown storage language, but not a deterministic pushdown storage language.

There is no deterministic pushdown storage automaton that accepts $\alpha\alpha^R$, because there is no center, and hence you do not know where to switch from storing to matching.
But there is a non-deterministic pushdown storage automaton accepting $\alpha\alpha^R$:

$V_I = \{a,b\}$
$V_O = \{Z_0,a,b\}$
$S = \{S_0,S_1\}$
$F = \{S_1\}$
$\delta = \{\delta_1,\ldots,\delta_7\}$

δ_1: $(a,S_0,e)\rightarrow(S_0,a)$
δ_2: $(b,S_0,e)\rightarrow(S_0,b)$
δ_3: $(a,S_0,a)\rightarrow(S_1,\sigma)$
δ_4: $(b,S_0,b)\rightarrow(S_1,\sigma)$
δ_5: $(a,S_1,a)\rightarrow(S_1,\sigma)$
δ_6: $(b,S_1,b)\rightarrow(S_1,\sigma)$
δ_7: $(e,S_1,Z_0)\rightarrow(S_1,\sigma)$

This automaton is non-deterministic since:
$(a,S_0,a)\rightarrow(S_0,a)$ and $(b,S_0,b)\rightarrow(S_0,b)$ are in δ^{CL}, and so are δ_3 and δ_4.

Compute: abba
Apply: δ_1, δ_2, δ_4, δ_5, δ_7, and you accept.

Compute: abab

a b a b e
↑
S_0
↓
Z_0

If we apply δ_3 we get:
a b a b e
↑
S1
↓

We are stuck here.
So we can only apply δ_1:
a b a b e
↑
S_0
↓
Z_0a

Apply δ_4:

a b a b e
↑
S_1
↓
Z_0

Now we can only apply δ_7 and get:

a b a b e
↑
S_1
↓

Once again, we are stuck.
Instead of applying δ_4 we could have applied δ_2:

a b a b e
↑
S_0
↓
Z_0ab

Here only δ_1 is possible, so we get:

a b a b e
↑
S_0
↓
Z_0aba

And here only δ_2 is possible, so we get:

a b a b e
↑
S_0
↓
Z_0abab

Now we are, once again, stuck.
We have gone though all the possibilities, hence abab is rejected.

A context free grammar is in **Greibach Normal Form** iff all rules are of the form:
$A \rightarrow a\alpha$, with $a \in V_T$ and $\alpha \in V_N^*$

Theorem: For every context free grammar there is an equivalent context free grammar in Greibach Normal Form.
Proof: Omitted

The fundamental theorem about pushdown storage languages is:

Theorem: The class of non-deterministic pushdown storage languages is exactly the class of context free languages.
Proof: Omitted

Both proofs are complex. The second theorem is standardly proved for context free grammars in Greibach Normal Form.

Without proof that it works, I *will* give here an algorithm for converting a context free grammar into an equivalent pushdown storage automaton.

Let $G = <V_N,V_T,S,R>$
M_G is given as follows:
$V_I = V_T$
$V_O = V \cup \{Z_0\}$
S_0 is the initial state, F is the final state.
$S = \{S_0,F\} \cup \{S_A: A \in V_N\} \cup X$
where *X* is as follows:
For each rule of the form $A \rightarrow \alpha$ in G, were $\alpha = \alpha_1 \ldots \alpha_n$, $\alpha_i \in V$
we have in *X* states: $X^{\alpha}_2,\ldots,X^{\alpha}_n$.
δ is given as follows:

Start: $(e,S_0,Z_0) \rightarrow (F,S)$

Push: For every rule $A \rightarrow \alpha$ in G with $\alpha = \alpha_1 \ldots \alpha_n$, $\alpha_i \in V$:

$(e,F,A) \rightarrow (S_A,\sigma)$

$(e,S_A,e) \rightarrow (X^{\alpha}_n,\alpha_n)$
$(e,X^{\alpha}_n,e) \rightarrow (X^{\alpha}_{n-1},\alpha_{n-1})$
…
$(e,X^{\alpha}_2,e) \rightarrow (F,\alpha_1)$

Pop: For every $a \in V_T$:

$(a,F,a) \rightarrow (F,\sigma)$

End: $(e,F,Z_0) \rightarrow (F,\sigma)$

The intuition is:
You read symbol a.
You look for a rule of the form $A \rightarrow a\ \alpha$.
You store α (it starts with a terminal or a non-terminal)
-if α starts with a_1 try to match input and store.
-if α starts with a non-terminal B look for a rule $B \rightarrow b\ \beta$ and push β onto the store.
At some point you get to a terminal c matching the right side of the store. You pop, go one level up, and try to match again.

Example: $a^n b^n$ $(n>0)$

G has rules: $S \rightarrow ab$, $S \rightarrow aSb$

$V_I = \{a,b\}$
$V_O = \{Z_0,a,b,S\}$
$S = \{S_0,F,S_S, X^{ab}_2, X^{aSb}_3, X^{aSb}_2\}$

Transitions:

Start: $(e,S_0,Z_0)\rightarrow(F,S)$

Push: $(e,F,S)\rightarrow(S_S,\sigma)$

$(e,S_S,e)\rightarrow(X^{ab}{}_2,b)$
$(e,X^{ab}{}_2,e)\rightarrow(F,a)$

$(e,S_S,e)\rightarrow(F^{aSb}{}_3,b)$
$(e,X^{aSb}{}_3,e)\rightarrow(X^{aSb}{}_2,S)$
$(e,X^{aSb}{}_2,e)\rightarrow(X,a)$

Pop: $(a,F,a)\rightarrow(F,\sigma)$
$(b,F,b)\rightarrow(F,\sigma)$

End: $(e,F,Z_0)\rightarrow(F,\sigma)$

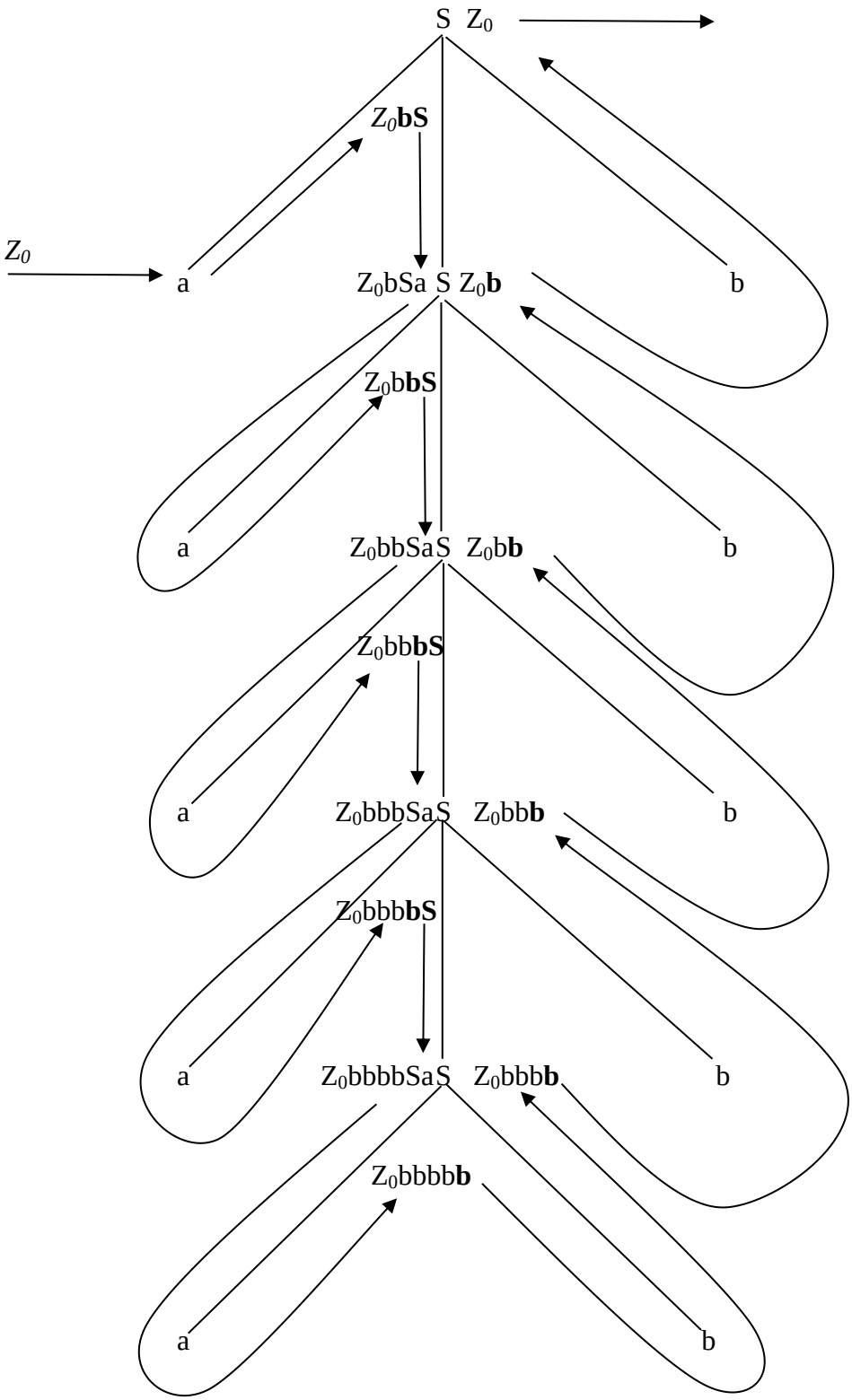
Compute: aabb

a a b b e	a a b b e	a a b b e	a a b b e	a a b b e	a a b b e
↑	↑	↑	↑	↑	↑
S_0	F	S_S	$X^{aSb}{}_3$	$X^{aSb}{}_2$	F
↓	↓	↓	↓	↓	↓
Z_0	Z_0S	Z_0	Z_0b	Z_0bS	Z_0bSa

a a b b e	a a b b e	a a b b e	a a b b e
↑	↑	↑	↑
F	S_S	$X^{ab}{}_2$	F
↓	↓	↓	↓
Z_0bS	Z_0b	Z_0bb	Z_0bba

a a b b e	a a b b e	a a b b e	a a b b e
↑	↑	↑	↑
F	F	F	F
↓	↓	↓	↓
Z_0bb	Z0b	Z_0	

We accept aabb.


S Z0
Z0bS
Z0
a
Z0bSa S Z0b
b
Z0bbS
a
Z0bbSaS Z0bb
b
Z0bbbS
a
Z0bbbSaS Z0bbb
b
Z0bbbbS
a
Z0bbbbSaS Z0bbbb
b
Z0bbbbb
a
b

Cartesian Product Automata

We come back to finite state automata.
Let M and N be two finite state automata.

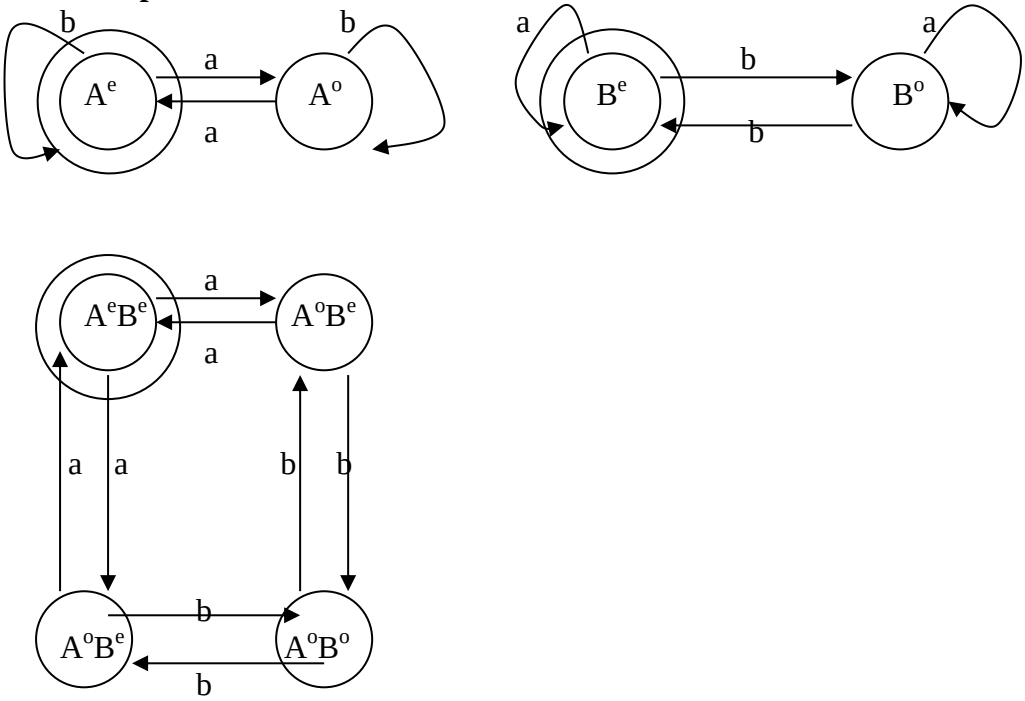
The **Cartesian Product Automaton M × N** is defined by:
1. $V_I = V_M \cup V_N$
2. $S_{M\times N} = S_M \times S_N$ (= {<A,B>: A ∈ S_M and B ∈ S_N}
3. $S_{0,M\times N} = <S_{0,M},S_{0,N}>$
4. $F_{M\times N} = \{<A,B>: A \in F_M \text{ and } B \in F_N\}$
5. $\delta(a,<A,B>) = <A',B'>$ iff
   $\delta(a,A)=A'$ and $\delta(a,B)=B'$

Fact: M × N is a finite state automaton and L(M × N)= L(M) ∩ L(N)
Proof: This is obvious from the construction.

This proves directly:

Corrollary: If A and B are regular, then A ∩ B is regular.

Example:



Now let M be a **pushdown storage automaton** and N a **finite state automaton**. Since M is itself a finite state automaton, we can define $M \times N$ just as above. It is the Cartesian product finite state automaton, with the store inherited from M. But $M \times N$ is, of course, itself a pushdown storage automaton. And it is easy to see which language it accepts: while a string is running through M, it runs simultaneously through N (where simultaneously means that the steps where the automata change state while progressing to reading the next symbol of the input are simultaneous). This means that of the strings accepted in a final state in M, the ones that are accepted by $M \times N$ are the ones that end up simultaneously accepted by N. Thus, $M \times N$ accepts $L(M) \cap L(N)$. This means that, as promised, we have proved:

Theorem: If A is context free and B is regular, then $A \cap B$ is context free.

Product automata, then, give us a simple way of writing a finite state automaton for the intersection of two regular languages that we have automata for, and a pushdown storage automaton for the intersection of a context free language and a regular language that we have automata for. As we have seen, the theorem does not extend to the intersection of two context free languages, and it's simple to see why the construction doesn't generalize: you can unproblematically take the products of the finite state parts of two pushdown storage automata, but you cannot necessarily imitate the effects of two pushdown stores by one store.

There is a good reason why you cannot do the latter. Imagine extending the concept of pushdown storage automata to what we might call **two track pushdown storage automata**:

> A **two track pushdown storage automaton** works exactly like a pushdown storage automaton, except that it has **two pushdown stores** A_1 and A_2, it will read a symbol on the input and each of the tops of A_1 and A_2 and it can decide to push down or pop on A_1 or on A_2 (or both).

Two track pushdown storage automata are clearly more powerful than pushdown storage automata. For instance, the language $a^n b^m c^n d^m$ is easily recognized: push the a's onto store A, and the b's onto store B, match c's with store A, and match d's with store B.

What are the languages recognized by two track pushdown storage automata?

Read the whole string onto store A. Do the rest with 0-moves.
-You can add as many symbols as you want at the end of the string by pushing symbols onto A.
-You can add as many symbols as you want at the beginning of the string, by moving the whole string symbol by symbol to store B and add symbols after them.
-You can go to any position in the string by moving symbols one by one to the other store until you reach the right one.
-This way, you can replace any symbol by another symbol in the string, by making the first the top of one store in the way described above, erazing it there and adding the other, and then move the symbols so as to bring us back to the original position.
These operations characterize Turing machings:

A **Turing machine** is a tuple $M = <S, \Sigma, V_0, \delta, S_0, F>$ with

S a finite set of states, V_0 the tape alphabet, $\Sigma \subseteq V_0$, the input alphabet, S_0 the initial state and $F \subseteq S$ the set of final states.

$\delta: S \times (V_0 \cup \{e\}) \rightarrow S \times (V_0 \cup \{e\}) \times \{L,R,N\}$ is a partial function

We have one two-way infinite tape. The input string α is written on the tape.

A **computation path for** α **in** M is a sequence of computation steps beginning in S_0 reading the first symbol of α on the tape, following instructions in δ until M halts.

A **computating path processing** α **in M** is a computation path for α in M which halts with the reading head on the input tape reading e **after** it has read all symbols in α.

$\alpha \in V_I^*$ is **accepted by** M iff there is a computation path processing α in M where at the end of the path **M is in a final state**.

The symbols L(eft), R(ight), N(euter) control the cursor direction:

$\delta(S_i,a) \rightarrow (S_j,b,L)$ means: on reading a on the tape in state S_i, go to state S_j, **replace** a by b, and move the cursor one position left.

Theorem: The languages recognized by Turing machines are exactly the type 0 languages.

Proof: Omitted.

Corrollary: The languages recognized by two track pushdown storage automata are exactly the type 0 languages.

This means, then, that in general you cannot collapse two pushdown storage tapes into one (otherwise you could reduce every Turing machine to a pushdown storage automaton, which, of course, you can't).

Fact: Type 0 grammars, Turing machines, Recursively enumerable functions, and (several more) all characterize the same set of functions (in our case, languages).

-All formalizations of the informal notion of algorithmic function coincide.
-No functions have ever been found that are intuitively algorithmic, but not in this class of functions.

Church's Thesis: these are all equivalent and adequate formalizations of the notion of algorithmic function, computable function.

Linear bounded automata

A **linear bounded automaton** is a turing machine M with a finite tape with non-erasable endmarkers **bot** and **top**.

This means that all the computation steps must be worked out between **bot** and **top** and the amount of space on the tape is given for M. This turns out to be equivalent to a Turing machine where the space used is restricted to being a linear function of the input (hence the name).

Theorem: The class of languages recognized by linear bounded automata is the class of context sensitive languages.
Proof: Omitted

Language L in alphabet A is **recursively enumerable** iff L has a type 0 grammar.
Language L in alphabet A is **recursive** iff both L and A*–L have a type 0 grammar.

Fact 1: There are languages that are not recursively enumerable (intractable)
Fact 2: There are recursive enumerable languages that are not recursive.
Fact 3: There are recurstive languages that are not context sensitive.
Fact 4: All context sensitive languages are recursive.

THE EMPTY STRING

Up to now we have been a bit pedantic about the empty string. One reason for this was to make sure that the grammar classes defined were inclusive (since you must do something special for context sensitive grammars). (A second reason was that I wanted you to do the exercises without using $A \rightarrow e$). But at this point we can relax and be more inclusive about what we call regular grammars or context free grammars. For this we mention two facts:

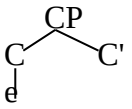
Fact: Let *G* be the class of grammars G with rules of the form $A \rightarrow \alpha B$, $A \rightarrow \alpha$, where $A,B \in V_N$ and $\alpha \in V_T^*$. Let L(*G*) be the class of languages determined by the grammars in G.
Then L(*G*) is the class of regular languages.

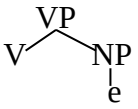
Hence, we can loosen up the format of right linear grammars to allow rules of the form $A \rightarrow e$ and also $A \rightarrow B$ (In the automata these correspond to empty-moves, moves labeled to e, and you can prove that automata with empty moves are equivalent to atomata without.)

Fact: Let *G* be the class of grammars G with rules of the form $A \rightarrow \alpha$, where $A \in V_N$ and $\alpha \in V^*$. Let L(*G*) be the class of languages determined by the grammars in G. Then L(*G*) is the class of contextfree languages.

This means that also for context free languages we can freely allow rules of the form $A \rightarrow e$. The reason is that if $e \notin L(G)$, any rule of the form $A \rightarrow e$ can be eliminated.

Instead of proving this, I indicate the trick:
In syntax we find empty categories, in parse trees that look like (1) and (2):

(1) 

or (2) 

We can eliminate these by encoding them on the higher CP and VP. We introduce two new non-terminals:

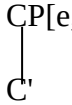
CP[e,L,C] and VP[e,R,NP]

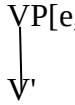
(They only **look** complex, but they are just A_{27} and B_{49}).

and two new rules:

$CP[e,L,C] \rightarrow C'$
$VP[e,R,NP] \rightarrow V'$

With this we generate parse trees:

(3) 

(4) 

The effect will be the same, but now we can eliminate the rules C→e and NP→e.

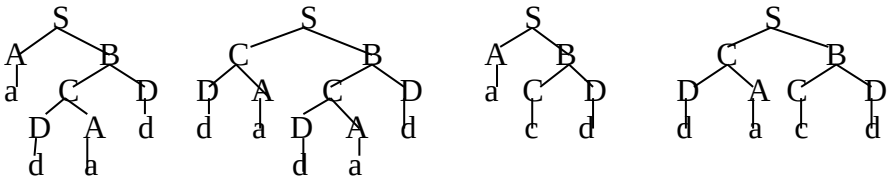
Vice versa, if we know that we can eliminate empty categories without affecting the generative power, we can also just introduce them without affecting the generative power. From now on we are not going to distinguish between the definition of context free grammars that allows empty rewriting and the definition that doesn't.

FEATURES AND CONTEXT FREE GRAMMARS

Take the following context sensitive grammar:

S→AB
S→CB
B→CD
C → DA
A→a
D→d
AC→c which we read as: <C→c,<A,e>>

This grammar generates four constituent structure trees, two in a context free way, and two in a context sensitive way:



Under what conditions can we rewrite C as c?
Answer: If C is in a configuration with the following properties:

1. You go up from C two nodes.
2. You go down there one node to the left.
3. If that node is A, or A occurs somewhere on the rightmost path down from that node, you can rewrite C as c.

This is a very context sensitive description, but does that mean that it is a context sensitive property? What do we mean by a context sensitive property? Roughly:

> **A property P of trees is type n** iff any grammar of type n can be turned into a grammar of type n which enforces the property on all its generated trees.
>
> Let's say that if a property of trees P is type n, P **can be encoded in type n grammars**.

So, the property 'c occurs in a tree as daughter of C iff C is in the above configuration in that tree' would be a context sensitive, rather than context free property, if you **need** a context sensitive grammar to enforce it, if it cannot be encoded in context free grammars.

On this analysis, the fact that we **formulate** the property in a context sensitive way means nothing: the question is whether we can or cannot **enforce** that property in context free grammars as well.

In general, the question what properties of trees can be encoded in what grammars is a highly non-trivial question. The point of this section is to show that context free grammars are capable of encoding far more properties than you might think.
Like the above property.

We can encode properties of trees in context free rules by using **features**.

Features are introduced in the context free rules. They can be manipulated in the grammar, but **only** locally, i.e. **per** rule.
That is, a context free rule constrains the relation between a node in a tree and its direct daughters. Since features are added to context free rules, they too can only constrain the relation between a node and its direct daughters.
Thus, we couldn't simply add a feature F and stipulate that it means: "in any constituent structure tree where this node occurs, there is a cut through the tree such that node A is directly left of this node in the cut."
We couldn't do this, because it isn't guarantees that we have a way in the grammar of **enforcing** the interpretation.

However, there are certain operations on features that we **can** define locally, and with that, there are properties of trees that we **can** enforce. Here are some basic operations: (my interpretation is bottom up, but that is inessential).

Feature generation:

X<A>→α	Interpretation: α introduces feature <A> on its mother X.
X<A>→Y A	Interpretation: right daughter A introduces feature <A> on its mother X.

Feature passing:

X<A>→Y<A> Z	Interpretation: feature <A> passes up from right daughter Y to mother X.
X<A>→Y Z<A>	Interpretation: feature <A> passes up from left daughter Z to mother X.

Feature checking:

X→Y<A> Z<A>	Interpretation: X is allowed if both daughters have feature A.
X→A Z<A>	Interpretation: X is allowed if the left daughter is A, and the right daughter has feature <A>.

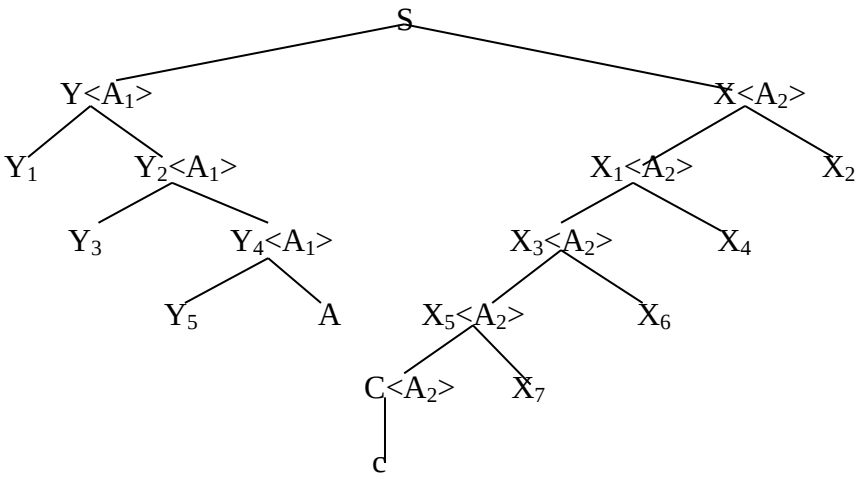
Feature checking and passing:

X<A>→Y<A> Z<A>	Interpretation: X is allowed if both daughters have feature <A>, and <A> is passed up.

We can use such feature systems to encode certain phenomena that look, at first sight, context sensitive.
-Let c introduce $<A_2>$ on C.
-pass $<A_2>$ up from C in rules where C is the left daughter.
-pass $<A_2>$ up from the left daughters until it is passed on to X, where X is the right daughter of S in rule S→Y X.
-Let A introduce feature $<A_1>$ on its mother in a rule where A is the right daughter.
-pass $<A_1>$ up from there from right daughters to mothers, until it is passed on to Y, where Y is the left daughter of S in rule S→ Y X.

-Require $<A_1>$ and $<A_2>$ to match there (= check that both features are present on the daughters of S).

A system of rules like this will enforce trees to look like:

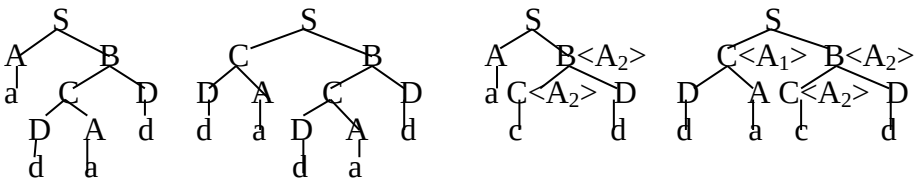


This will **enforce** that if c occurs in the tree as daughter of C, there is a cut in the tree with A directly left of C in the cut.

In the case of the little grammar we gave, we encode it in the following rules:

$S \rightarrow AB$, $S \rightarrow CB$, $B \rightarrow CD$, $C \rightarrow DA$, $A \rightarrow a$, $D \rightarrow d$
$S \rightarrow A\ B\langle A_2\rangle$
$S \rightarrow C\langle A_1\rangle\ B\langle A_2\rangle$
$B\langle A_2\rangle \rightarrow C\langle A_2\rangle\ D$
$C\langle A_2\rangle \rightarrow c$
$C\langle A_1\rangle \rightarrow D\ A$

This grammar generates the following trees:



But, of course, **there is nothing context sensitive** about this.
We have just added five new context free rules to the grammar involving three new non-terminal symbols $C\langle A_1\rangle$, $C\langle A_2\rangle$, $B\langle A_2\rangle$. The resulting grammar is, of course, perfectly context free.

This means that, as long as we add a finite number of context free rules, mentioning a finite number of feature-non-terminals, the grammar stays context free. And this means that we can encode indeed far more properties than we would have thought at first sight. So, far more phenomena that, at first sight, we might think require context sensitive rules, turn out to be perfectly context free.
For instance, if you make sure that feature $<A_1>$ is only passed up from right most daughters, and feature $<A_2>$ is only passed up from left most daughters, and that any rule that mentions both $<A_1>$ and $<A_2>$, mentions them on **adjacent** daughters, then it doesn't matter **how deep** A_1 and A_2 are introduced: the nodes where they are introduced will be sitting next to each other on some cut through the tree.
Hence, we can encode this seeming context sensitive property with a finite number of features that are only intruduced locally in context free rules. Hence the property is context free.

Of course, whether the mechanisms described as feature introduction, feature passing, and feature checking really **mean** that, depends on the grammar. The point is, that you can easily **set up** the grammar in such a way that that interpretation is enforced. For checking, this means, say, allowing only rule $X \rightarrow Y<A>\ Z<A>$, but not $X \rightarrow Y\ Z<A>$ and not $X \rightarrow Y<A>\ Z$.
But that is only a matter of **your** proficiency in writing context free grammars.

Example of the use of features:
Number in Espresso:

$D<plur> \rightarrow$ dis
$N<plur> \rightarrow$ manis
$A<plur> \rightarrow$ altis
$V<plur> \rightarrow$ lopetis
$V<plur> \rightarrow$ kusetis
etc.

Rules with features:

$N<\alpha> \rightarrow A<\alpha>\ N<\alpha>$
$NP<\alpha> \rightarrow D<\alpha>\ N<\alpha>$
$VP<\alpha> \rightarrow V<\alpha>$
$VP<\alpha> \rightarrow V<\alpha>\ NP<\beta>$
$S \rightarrow NP<\alpha>\ VP<\alpha>$

where $\alpha,\beta \in \{sing, plur\}$

If we deal both with gender and number, we get category labels like N<plur><fem>. The complexity is only a matter of notation, we could just as well have chosen new label N_{27} for this, or, for that matter, Z. The context freeness is not affected.
But, of course, it is useful to have features in the grammar, because it allows us to write one rule schema summarizing lots of instances, and it allows us to separate out the categorial restrictions (an NP combines with a VP to give a sentence) from agreement restrictions: an adjective agrees with a noun, a subject agrees with the verb.

A more common format for feature grammars is to regard a node in a tree a **feature matrix:** a set of function-value pairs like:

$$\begin{pmatrix} \text{CATEGORY: D} \\ \text{NUMBER: plur} \\ \text{GENDER: male} \end{pmatrix}$$

A grammar generating trees with node labels of this form is called an **attribute grammar**.
In the most general form of attribute grammars, you can allow **complex values**. For instance, you could have a node saying:

[GAP-RECONSTRUCTION: T_i]

where T_i is a tree, interpreted as an instruction to attach at this node a subtree $T_{i.}$. Unrestricted attribute grammars are equivalent to type 0 grammars. Attribute grammars that are a bit richer than context free grammars are used in HPSG.

To give linguistic bite to all this, I will discuss wh-movement in English.

Example: wh-movement.

I am interested here in generating trees that you may be familiar with from the syntactic literature. I will be interested in accounting for the following data in English (chosen in such a way so as not to have to deal with **everything** under the sun).

(1) a. John knew that Mary knew that John kissed Mary.
b. John knew that Mary knew e John kissed Mary.
c. John knew e Mary knew that John kissed Mary.
d. John knew e Mary knew e John kissed Mary.

Complementizer *that* is optional with the verb *know*.

(2) a. *John knew whom that Mary knew that John kissed e.
b. *John knew whom that Mary knew e John kissed e.
c. John knew whom e Mary knew that John kissed e.
d. John knew whom e Mary knew e John kissed e.

The wh-phrase occurs in the higher position instead of as complement of the lower verb, but can show the case it would have if it had been the complement of the lower verb.
The wh-phrase requires the complementizer next to it to be empty, but, in extraction from object position, there is no such requirement on lower complementizers.

(3) a. *John knew who that Mary knew that e kissed Mary.
b. *John knew who e Mary knew that e kissed Mary.
c. *John knew who that Mary knew e e kissed Mary.
d. John knew who e Mary knew e e kissed Mary.

When the extraction is from subject position, we find a further requirement, the that-trace effect: extraction from subject requires the next complementizer up to be empty as well (as shown by the contrast between 3b and 3d).

I let the wh-phrase land under *knew*, because I don't want to deal with inversion.

The informal description here is highly context sensitive:
-Movement is a restructering operation on trees, relating an empty position as far down as you want in the tree to a subtree higher up.
-We notice long distance case-agreement: whether you can get *whom* in the higher position depends on where, deep down, the trace is.

But is wh-movement context sensitive?
Answer: not as far as **these** data are concerned, because these operations can be fully encoded with features on context free rules.

I will start by assuming a set of nine feature labels:
{e,i,wh,NP,nom,obj,bot,mid,top}.
My set of features will be a finite set of sequences of these feature labels.

I will specify the interpretations to be enforced of the sequences I specify, on the understanding that this interpretation of a sequence is to be carried over in longer sequences.

Interpretations:
<i,wh,NP,nom> on node A indicates:
A is part of a wh-chain i with bottom an NP node with nominatice case.

<i,wh,NP,obj> on node A indicates:
A is part of a wh-chain i with bottom an NP node with objective case.

<i,bot,wh> on node A indicates:
A is empty and the bottom of a wh-chain i. (bottom trace)

<i,mid,wh> on node A indicates:
A is empty and an intermediate node of a wh-chain i. (intermediate trace)

<i,top,wh> on node C' indicates:
The top of wh-chain i is going to be a left sister of this node.

<i,top,wh> on node NP indicates:
This node is the top of wh-chain i.

<e> on node C indicates:
A is an empty complementizer.

As usual, these interpretations mean nothing if we cannot enforce them. The point of the example is to show that we can indeed easily enforce these interpretations. We do that by specifying the rules.

Let $\alpha \in \{\alpha_1,\alpha_2\}$, α_1=<i,wh,NP,nom>, α_2=<i,wh,NP,obj>

$V_1 \rightarrow$kissed $V_2 \rightarrow$knew
C$\rightarrow$that C<e>$\rightarrow$e
NP$\rightarrow$John NP$\rightarrow$Mary NP<e>$\rightarrow$e

Traces
NP<bot,i,wh>$\rightarrow$e NP<mid,i,wh>$\rightarrow$e
Bottom trace Intermediate trace

Wh-expressions
NP<top,α_1>$\rightarrow$who NP<top,α_2>$\rightarrow$whom

(While this is written as a pair, we assume, as usual, that sequence formation is associative, so this are really quintuples.)

Verb phrases

$VP \rightarrow V_1$ NP $VP \rightarrow V_2$ CP

$VP\alpha \rightarrow V_2$ CPα (pass α up)

$VP\alpha_2 \rightarrow V_1$ NP<bot,i,wh> (a bottom trace in the object position of VP introduces α_2 on the VP)

Sentences

$S \rightarrow$ NP VP

$S\alpha \rightarrow$ NP VPα (pass α up)

S<bot,α_1>$\rightarrow$NP<bot,i,wh> VP (a bottom trace in the subject positon of S introduces α_1 on S, **and** introduces bot on S to trigger the that-trace effect)

CPs

$CP \rightarrow$NP<e> C' (for CPs where nothing lands)

$CP\alpha \rightarrow$NP<mid,i,wh> C'α (pass α up and leave an intermediate trace)

$CP \rightarrow$NP<top,α> C'<top,α> (introduce the top of the wh-chain i)

C's

$C' \rightarrow$ C S

$C'\alpha \rightarrow$ C Sα (pass α up)

$C' \rightarrow$ C<e> S

$C'\alpha \rightarrow$ C<e> Sα (pass α up)

$C'\alpha \rightarrow$ C<e> S<bot,α> (pass α up with that-trace effect)

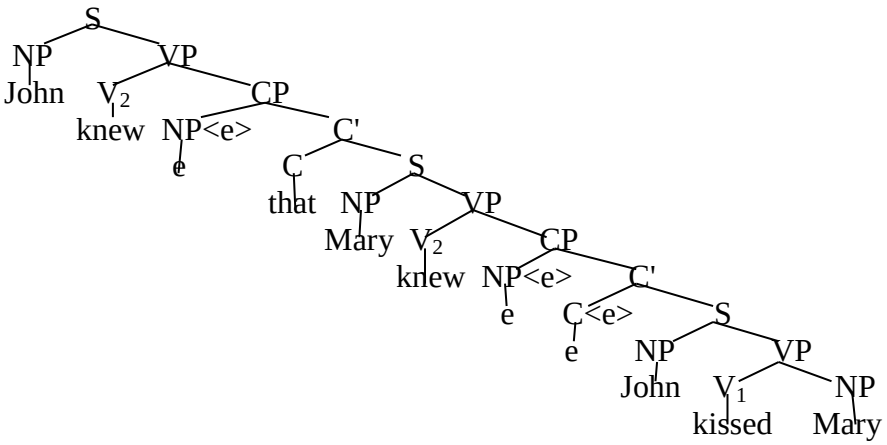
C'<top,α>$\rightarrow$C<e> Sα (prepare for introducing the top of the chain)

C'<top,α>$\rightarrow$C<e> S<bot,α> (prepare for introducing the top of the chain first time round)

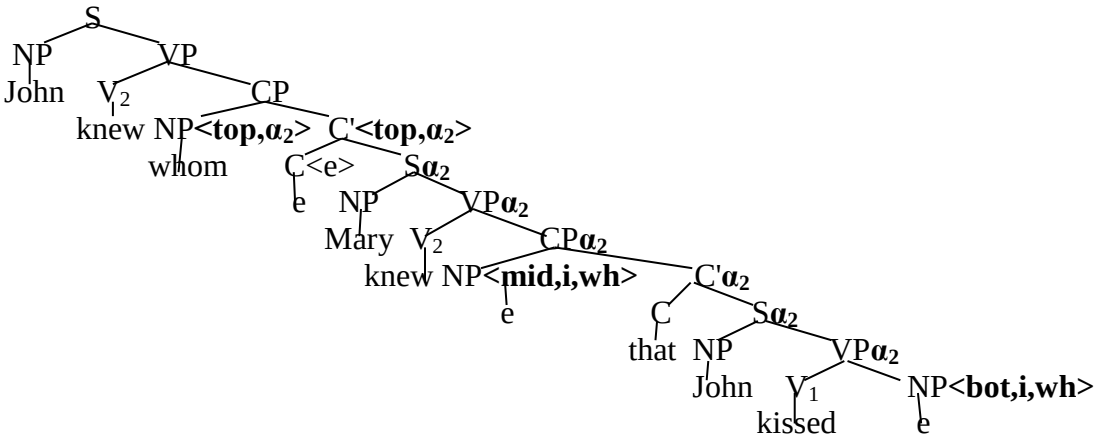
Important in this format:
Categories with features behave like independent categories. So if we find on a node label Sα, we **cannot** apply rule $S \rightarrow$NP VP to this node. We **must** look for a node with label Sα on the left.

Sample derivations:

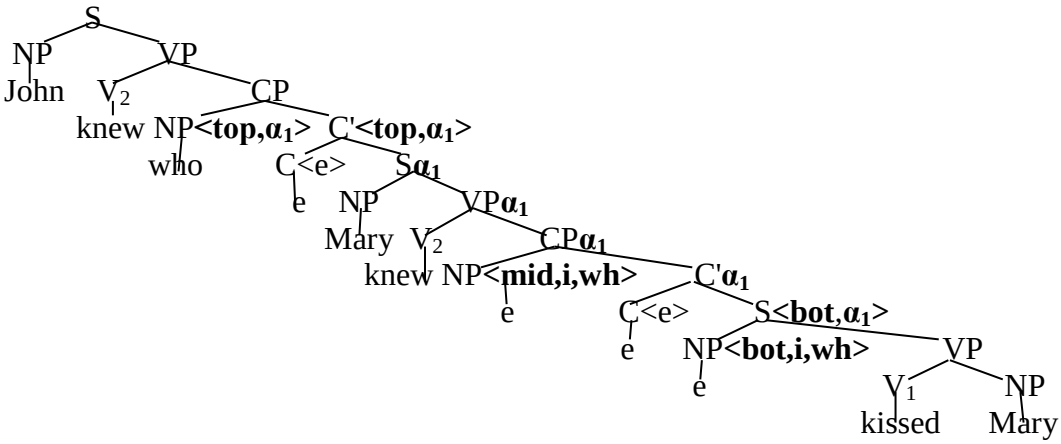
1d. John knew that Mary knew John kissed Mary.



2c. John knew whom Mary knew that John kissed.



3.d John knew who Mary knew kissed Mary.



The grammar we have given is perfectly context free, and more than just 'descriptively adequate'. We **encode** in the feature passing mechanism the very same notions of chain that 'real' movement theories assume. This means that the very same long distance relations between nodes that the movement theory assumes are de facto encoded in the feature passing mechanism. This means that we do not just describe the same set of facts in a context free way, but we have given a context free reduction of the **linguistics anaysis**: the linguistic analysis of the wh-movement facts commonly adopted does not **essentially** rely on non-contextfree properties and relations of trees. In other words, these wh-movement facts do not only not show that the string set of English is not context free, but stronger, the **analytic tools** commonly assumed (movement and chains) are **perfectly normal context free tools**.

As far as I am concerned, this doesn't mean that you **must** use the feature passing reconstruction rather than chains and movements. We use whatever helps us express the generalizations we want to cover the best. But it is useful to know that, when needed, we can convert this part of the theory to a context free format (and use, for instance, the facts known about context free parsing).

The feature passing mechanism has actually some direct advantages too.
As the name already suggests, 'across the board' movement has always been some of an embarrasment for the classical movement account. As is well known, wh-chains can have more than one tail, if the tails come from each conjunct in a conjunction:

(4) a. John knows whom$_i$ Mary thinks that Bill kissed e_i and Henry likes e_i.
b. *John knows whom$_i$ Mary thinks that Bill kissed e_i and Henry likes Susan.
c. * John knows whom$_i$ Mary thinks that Bill kissed Susan and Henry likes e_i.

This is a bit of an embarrasment for the classical theory, because the *whom* is moved from a conjunction, which is not supposed to be possible, and how **can** you move one thing from two places simultaneously. Clearly, the **literal** movement interpretation is under stress in these cases.
But it is very easy to modify the feature passing analysis to fit across the board movement facts: allow α to be check and passed on from the conjuncts to a conjunction, α is passed on to a conjunction, and higher up from there if **both** conjuncts have α.

There is one aspect of the analysis that requires further investigation, and that is **index** i marking wh-chain i.
Since there can be arbitrarily many wh-chains in a sentence, we need to worry about the question of how many indices we must require in the grammar.

The fact is, that there is no problem if there are no nodes that are part of more than one wh-chain. In that case, we can just use one and the same index (or rather, we don't need an index) for both chains: a node is part of the wh-chain determined by the closest top and closest bot, and this can be unambiguously determined for each node, even if there are 5000 non-overlapping wh-chains in the sentence.
But matters are different, if chains are allowed to overlap.

It so happens that this is not a problem for English, because, as is well known, English doesn't allow overlapping wh-chains, i.e, you can't have the following:

(5) a. *John knows who_i Mary knows $whom_j$ Bill thinks e_i loves e_j
b. *John knows $whom_i$ Mary knows who_j Bill thinks e_i loves e_j

But other languages seem to allow this (Engdahl 1986 mentions Swedish).
Let's think about wh-chains that originate from the same sentence. If we only need to take into account the arguments of the verb, then there still isn't a problem: verbs have only a finite number of arguments (at most 3), so 3 wh-indices would be enough, if each argument of one verb can be the bottom of a wh-chain. But if the language allows preposition stranding, and simultaneous extraction, then there might be a problem. Because then you could get something like the following:

who_1....who_2.....who_4...who_3... ($e_1$verb e_2 in e_3 for e_4 ...)

If we can add as many prepositional adjuncts as we want, we could, theoretically, have as many simultaneous extractions as we want.
But that **might** mean that we might have to be able to distinguish arbitrarily many wh-chains on the same category label, and that **might** mean that we would need arbitrarily many indices. Since our features are by necessity finite, we couldn't do that with features and our analysis might well go beyond context free (but the argument must be tied to the exact nature of the data. As we will see, non-context freeness arguments are rarely straightforward).

What happens if we allow an infinite set of indices in a context free grammar?
Well, that depends, of course, on what you do with them. This question has been studied most extensively in the study of the socalled **indexed languages**.

INDEXED GRAMMARS

Indexed grammars are a generalization of context free grammars.

An **indexed grammar** is a tuple $G = <V_T, V_N, I, S, R>$ where:

1. V_N, V_T, S are as usual.
2. I is a finite set of indices.
3. R is a finite set of rules of one of the three following forms:

a. $A \rightarrow \alpha$ where $A \in V_N$, $\alpha \in V^*$.

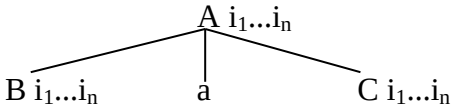
b. $A \rightarrow Bf$ where $A, B \in V_N$, $f \in I$.

c. $Af \rightarrow \alpha$ where $A \in V_N$, $\alpha \in V^*$, $f \in I$.

This looks much like context free grammars. The difference comes in the derivation.
-In the parse trees of an indexed grammar, we associate with every node with a non-terminal label a string of indices. So node labels are of the form: $A\ i_1...i_n$.
-The rules are interpreted as follows:

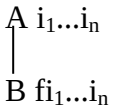
a. If G contains $A \rightarrow \alpha$ and T is a parse tree for G with leaf node n with label $A\ \delta$ we can expand node n in the following way:
-add the symbols in α left to right as daughternodes to node n (same as for context free grammars)
-copy index δ to every daughter of n which has a non-terminal label.

Example: rule $A \rightarrow BaC$ and node $A\ i_1...i_n$ give tree:



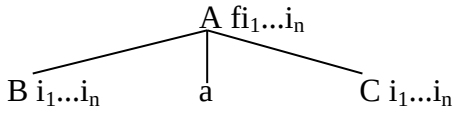
b. If G contains $A \rightarrow B\ f$ and T is a parse tree for G with leaf node n with label $A\ \delta$ we can expand node n in the following way:
-add a daughternode with label B to node n (same as for context free grammars)
-add f to the top of the string δ on the daughter (i.e. the daughter has label B f ^ δ.

Example: rule $A \rightarrow B\ f$ and node $A\ i_n...i_n$ give tree:



c. If G contains $A\ f \rightarrow \alpha$ and T is a parse tree for G with leaf node n with label $A\ f\hat{}\delta$ we can expand node n in the following way:
-add the symbols in α as daughter nodes to n in left right order (same as for context free grammars).
-copy index δ to every node of n which has a non-terminal label.

Example: rule A f $\rightarrow$ B a C and node A $fi_1...i_n$ give tree:



So each node in a parse tree carries either a terminal label or a non-terminal label and a pushdown store, a stack, of indices. In the rules of type a, the index stack gets copied to all the non-terminal daughters; in the rules of type b, we push an index on top of the index stack of the daughter; in rules of type c, we copy the index stack to all the non-terminal daughters, while removing an index from the top of each of these stacks.
The remaining notions are the same as for contextfree grammars.
We call the languages generated by indexed grammars **indexed languages**.

Fact: Every context free grammar is an indexed grammar.
Hence the class of context free languages is contained in the class of indexed languages.

This is obvious from the definition: restrict yourself to rules of type a only, and you have a context free grammar.

Fact: The class of indexed languages is contained in the class of context sensitive languages.

This was proved in Aho 1968, who introduced indexed grammars. The containment is proper: there are context sensitive languages which are not indexed languages.
For instance:
for n>0: $n! = 1 \times ... \times n$
The language: $a^{n!}$ (n>0) is context sensitive, but not an indexed language.
More natural, the language MIX = $\{\alpha \in \{a,b,c\}^*: |a|_\alpha = |b|_\alpha = |c|_\alpha\}$ is context sensitive, and in the 1980ies Bill Marsh conjectured it to be not an indexed language. At present this conjecture has still not been proved (which gives us an Argument by Intimidation that it isn't an indexed language, since really smart people have been unable to come up with an indexed grammar for it.)

Fact: There are indexed languages that are not context free.

We are going to show that ourselves, by showing that the languages that earlier we proved to be not context free all are indexed languages.

In the examples to follow we use two indices f,g in which g plays the role of marking the bottom of the index stack. (This is not part of the definition, but encoded in the grammar.)

Example: **$a^n b^n c^n$ (n>0) is an indexed language.**

V_N = {S,T,A,B,C}

V_T = {a,b,c}

I = {f,g}

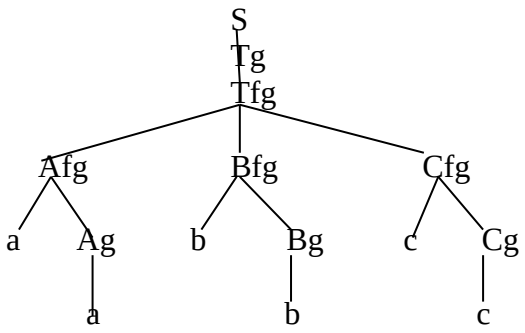
R:

$S \rightarrow T\ g$	$A\ f \rightarrow a\ A$	$A\ g \rightarrow a$
$T \rightarrow T\ f$	$B\ f \rightarrow b\ B$	$B\ g \rightarrow b$
$T \rightarrow A\ B\ C$	$C\ f \rightarrow c\ C$	$C\ g \rightarrow c$

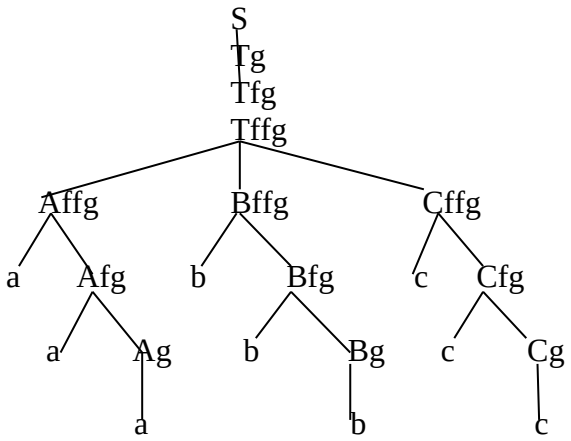
The trick in this grammar, and in indexed grammars in general, is that you start with a routine of stocking up as many indices f as you want (starting with g) on a single node T, then you spread them to the daughers A, B and C, and then you continue, for each of these nodes, with a routine of eating the indices up.
What you can do in this way, which you can't do in context free grammars, is let the length of the top stocking up tree segment control the length of the subtrees dominated by A and by B and by C.

Sample derivations:

aabbcc:



aaabbbccc:



Example: a^{2^n} **is an indexed language.**

$V_N = \{S,A,B\}$

$V_T = \{a\}$

$I = \{f,g\}$

R:

$S \rightarrow A\ g$

$A \rightarrow A\ f$

$A\ f \rightarrow B\ B$

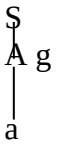
$B\ f \rightarrow B\ B$

$A\ g \rightarrow a$

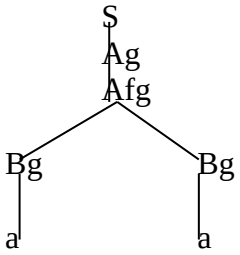
$B\ g \rightarrow a$

Sample derivations:

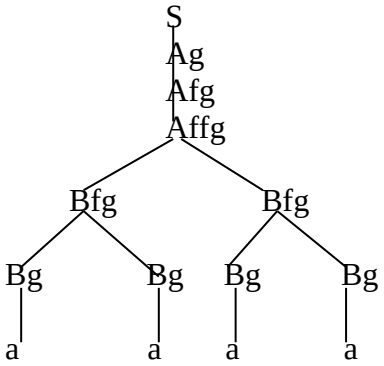
a:



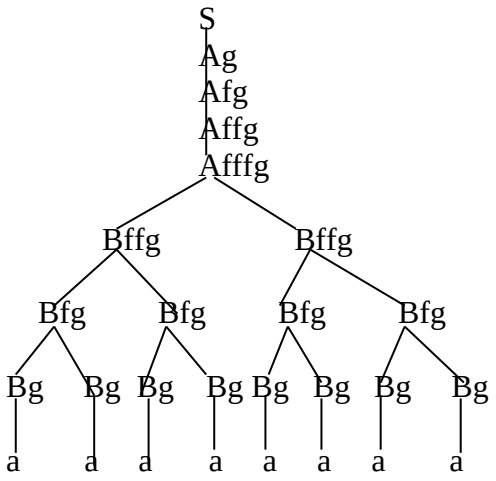
aa:



aaaa:



aaaaaaaa:



Example: a^{n^2} (n>0) **is an indexed language.**

V_N = {S,A,B,C,D}

V_T = {a}

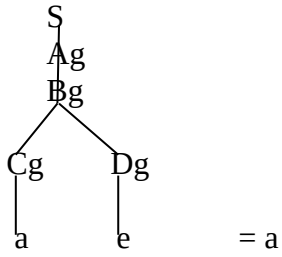
I = {f,g}

R:

S→ A g	D f→B
A→ A f	D g→e
A→B	C f→aaC
B→C D	C g→a

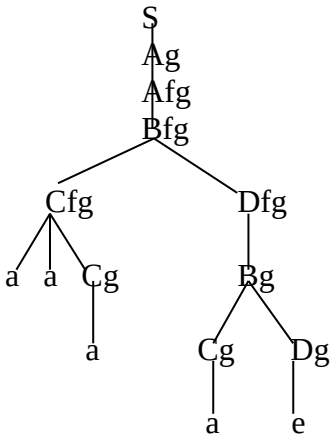
Sample derivations:

a:



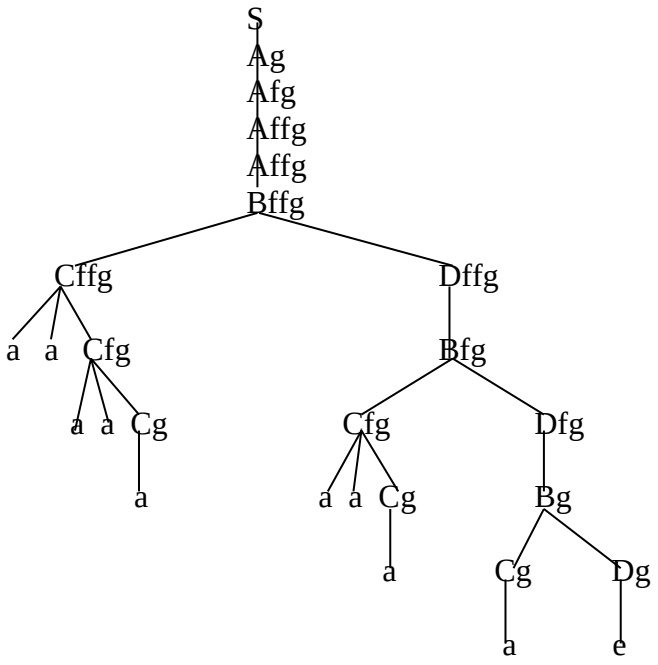
= a

aaaa:



= aaaa

aaaaaaaaa:



= aaaaaaaaa

One more fact:

Fact: The class of languages generated by **right linear indexed grammars** is exactly the class of context free languages.

Proof:
This is obvious. The right linear grammar forms a finite state automaton. The index is a pushdown store. In a right linear index grammar you have in every parse tree only one spine of non-terminals on the right, so you can only push and pop on the pushdown store, the stores don't spread. Clearly, this is just an alternative description of a pushdown storage automaton.