FORMAL LANGUAGES

Fred Landman

Class Notes, revised 2017

INTRODUCTION

Grammar: structure or rule-system which describes/explains the phenomena of a language, constructions.

-Motivated by linguistic concerns.
-But also by computational concerns.

-a grammar supports the judgements about the data on the basis of which it is built.
-processing of sentences is rapid, online.

Suppose you have found a 'super grammar' for English, it works really well.
And then some nasty mathematician proves that your grammar cannot support a mechanism for **making** the judgements that it is based on.
Or some nasty mathematician proves that your grammar cannot support an online processing mechanism to process online even the sentences that we know we process online.
In that case, you have a super grammar, but apparently not the one that native speakers of English use, because they **do** have the judgements they do, and they **do** process online what they process online.

These are questions about the **power** of grammatical operations.
If the grammar doesn't have enough power, you can't describe what you want to describe.
If the grammar has to much power, it becomes a system like arithmetics: you can learn to do the tricks, but clearly native speakers do not have a lot of **native** judgements about the outcome of arithmetical operations, nor are they **natively** very good at doing arithmetics online. And, we know that there are good reasons why this is so: Arithmetics is a very powerful system, provably too complex to support systematically native judgements about the outcomes of the operations, and provably to complex to do online (though bits of it can be done very fast by pocket calculators, as we know).

So the queston is: How much power is needed, and how much power is too much?

Formal language theory provides a framework for comparing different grammatical theories with respect to power.

PART 1. STRINGS, TREES AND REWRITE GRAMMARS.

ALPHABETS, STRINGS, AND LANGUAGES

Let A be a set.
The set of **strings on A** is the set:

$A^* = \{\langle a_1,\ldots,a_n\rangle : n \geq 0 \text{ and } a_1,\ldots,a_n \in A\}$

The operation * is called the **Kleene closure**.

Concatenation, ^, is a two-place operation on A*:

$\langle a_1,\ldots,a_n\rangle \,\hat{}\, \langle b_1,\ldots,b_m\rangle = \langle a_1,\ldots,a_n,b_1,\ldots,b_m\rangle$

-<>, the **empty string,** is in A*, we write **e** for the empty string.

Another characterization of A*:

$A^* = \{\alpha$: for some $\alpha_1 \ldots \alpha_n \in A$ such that $n \geq 0$: $\alpha = \alpha_1 \hat{} \ldots \hat{} \alpha_n\}$

$$A^+ = \begin{cases} A^* & \text{if } e \in A \\ A^* - \{e\} & \text{if } e \notin A \end{cases}$$

Facts: - ^ is associative, but not commutative.

$(\alpha \,\hat{}\, \beta) \,\hat{}\, \gamma = \alpha \,\hat{}\, (\beta \,\hat{}\, \gamma)$

but not: $\alpha \,\hat{}\, \beta = \beta \,\hat{}\, \alpha$

- e is an identity element for ^:

for every $\alpha \in A^*$: $e \,\hat{}\, \alpha = \alpha \,\hat{}\, e = e$

This means that we understand, the bracket notation in such a way that, say, <a,e,b> is a pair (and not a triple).

Notation: we write $a_1 \ldots a_n$ for $\langle a_1,\ldots,a_n\rangle$, hence we also identify <a> and a.
Note that we can write, when we want to, ab as aeb or as eeeaeebeee.
This is the same string, by the fact that e is an identity element.

Example:
A = {a}
A* = {e, a, aa, aaa, aaaa, …}
A^+ = {a, aa, aaa, aaa, …}

If α is $a_1 \ldots b_1 \ldots b_n \ldots a_m$, we call $b_1 \ldots b_m$ a **substring of** α.
A **prefix** of α is an **initial substring of** α, a **suffix** of α is a **final substring of** α.
Fact: e is a substring of every string.

α is an **atom** in A* if $\alpha \neq e$ and the only substrings of α in A* are e and α itself.

An **alphabet** is a set A such that every element $a \in A$ is an atom in A*.

We call the elements of alphabet A **symbols** or **lexical items.**

Fact: If A is an alphabet, then every string in A* has a **unique** decomposition into symbols of A.

Example: {a, b, ab} would not be an alphabet, since ab is in {a,b,ab}, but it is not an atom. We restrict ourselves to alphabets, because then we can define the length of a string:

> Let A be an alphabet.
> The **length of string** α in A*, $|\alpha|$, is the number of occurrences of symbols of A in α.
> Let $a \in A$.
> $|a|_\alpha$ is the number of occurrences of symbol a in α.

Note that we do not allow the empty string to occur as a symbol in alphabet A. (This means that, for **alphabet** A, $A^+ = A^* - \{e\}$.)

Note further that in calculating the length of a string, we do not count e:
if A = {a,b}, |aaebbeaa| = |aabbaa| = 6

> A **language in alphabet** A is a set of strings in A*:
> L is a language in alphabet A iff $L \subseteq A^*$

Note: the theory is typically developed for languages in **finite alphabets** (important: this does **not** mean finite languages). That is, the lexicon is assumed to be finite.

In linguistics, the usual assumption is that the lexicon is not finite (missile, antimissile, antiantimissile,…). This is mainly a problem of terminology: the **formal** notion of grammar that we will define here will not distinguish between lexical rules and syntactic rules (but you can easily **introduce** that distinction when wanted). So, if the grammar contains lexical rules, the **alphabet** would be the finite starting set of irreducible lexical items.

TREES

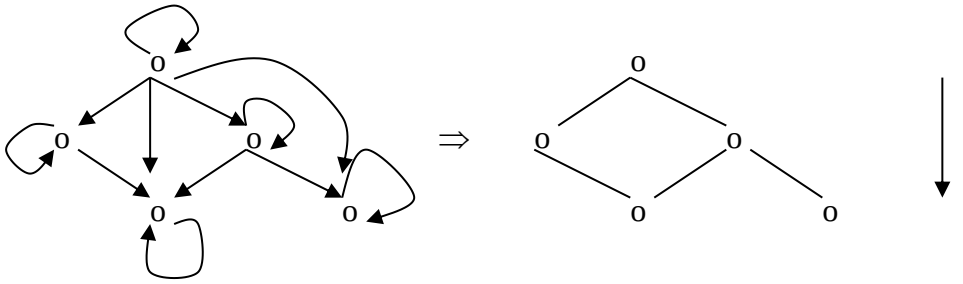
A **partial order** is a pair <A, ≤>, where A is a non-empty set and ≤ is a reflexive, transitive, antisymmetric relation on A

Reflexive: for all a ∈ A: a ≤ a
Transitive: for all a,b,c ∈ A: if a ≤ b and b ≤ c then a ≤ c
Antisymmetric: for all a,b ∈ A: if a ≤ b and b ≤ a then a=b

A **strict partial order** is a pair <A,<>, where A is a non-empty set and < is an irreflexive, transitive, asymmetric relation of A.

Irreflexive: for all a ∈ A: ¬(a ≤ a)
Antisymmetric: for all a,b ∈ A: if a < b then ¬(b < a)

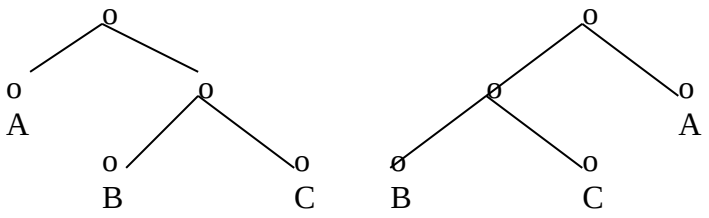
Graphs of partial orders:
-We don't distinguish between reflexive and irreflexive.
-We don't write transitivity arrows.
-The direction of the graph is understood.



A **tree** is a structure <A,≤,O,L> where:

1. A is a finite set of **nodes**.
2. ≤, the relation of **dominance**, is a partial order on A.
3. 0, the **topnode** or **origin** is the **minimum** in ≤:
   for every a ∈ A: 0 ≤ a
4. **Non-branching upwards**:
   For every a,b,c ∈ A: if b ≤ a and c ≤ a then b ≤ c or c ≤ b
5. ---

This is the standard notion of tree in mathematics. For linguistic purposes, we are interested in something more restricted. Up to now the following trees are exactly the same tree:



We want to distinguish these, and do that by adding a **leftness relation** L:

5. L , the **leftness relation** is a strict partial order on A satisfying:
   Semi-connectedness:
   for all a,b $\in$ A: $\neg(a \leq b)$ and $\neg(b \leq a)$ iff L(a,b) or L(b,a)

Fact: L satisfies **monotonicity**:
If $a \leq a_1$ and $b \leq b_1$ and L(a,b), then $L(a_1,b_1)$

Proof:
1. If $a \leq a_1$ and $b \leq b_1$ and L(a,b), then $L(a_1,b_1)$ or $L(b_1,a_1)$

Assume $a \leq a_1$ and $b \leq b_1$ and L(a,b).
-Assume $a_1 \leq b_1$. Then, by transitivity of $\leq$, $a \leq b_1$. Since $b \leq b_1$, it follows by non-branching upward that $a \leq b$ or $b \leq a$. Then $\neg$L(a,b). Contradiction.
Hence $\neg(a_1 \leq b_1)$.
-Assume $b_1 \leq a_1$. Then, by transitivity of $\leq$, $b \leq a_1$. Then $a \leq a_1$ and $b \leq a_1$, and by non-branching upward, $a \leq b$ or $b \leq a$. Then $\neg$L(a,b). Contradiction.
Hence, $\neg(b_1 \leq a_1)$.
By semi-connectedness, it follows that: $L(a_1,b_1)$ or $L(b_1,a_1)$.

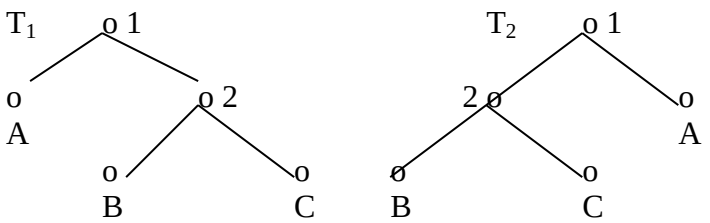
2. If $a \leq a_1$ and $b \leq b_1$ and L(a,b), then $L(a_1,b_1)$

-Assume $a_1 \leq b$. Then, by transitivity of $\leq$, $a \leq b$. Contradiction.
Hence $\neg(a_1 \leq b)$.
-Assume $b \leq a_1$. Then $a \leq a_1$ and $b \leq a_1$ and, by non-branching upwards, $a \leq b$ or $b \leq a$. Contradiction.
Hence $\neg(b \leq a_1)$.
Hence, it follows, by semi-connectedness, that $L(a_1,b)$ or $L(b,a_1)$.

Assume $L(b,a_1)$. Then L(a,b) and $L(b,a_1)$, and, by transitivity of L, $L(a,a_1)$.
This contradicts the assumption that $a \leq a_1$, so we have shown that $\neg L(b,a_1)$.
It follows indeed that $L(a_1,b)$.

Assume, $L(b_1,a_1)$. Since we have just shown that $L(a_1,b)$, it follows, by transitivity of L, that $L(b_1,b)$. Contradiction.
Hence $\neg L(b_1,a_1)$.
By (1) it now follows that: $L(a_1,b_1)$.

We make the convention of leaving out Leftness from the pictures of trees, and assume it to be understood as left in the picture.



Thus the picture T_1 summarizes the tree
$T_1 = \langle A,\leq,0,L_1\rangle$, where:
$A = \{1,2,A,B,C\}$
$\leq = \{\langle 1,1\rangle, \langle 1,A\rangle, \langle 1,2\rangle, \langle 1,B\rangle, \langle 1,C\rangle, \langle A,A\rangle,$
$\langle 2,2\rangle, \langle 2,B\rangle, \langle 2,C\rangle, \langle B,B\rangle, \langle C,C\rangle\}$
$0_{T1} = 1$
$L_1 = \{\langle A,2\rangle, \langle A,B\rangle, \langle A,C\rangle, \langle B,C\rangle\}$

and the picture T_2 summarizes the tree
$T_2 = \langle A,\leq,0,L_2\rangle$, where:
$A = \{1,2,A,B,C\}$
$\leq = \{\langle 1,1\rangle, \langle 1,A\rangle, \langle 1,2\rangle, \langle 1,B\rangle, \langle 1,C\rangle, \langle A,A\rangle,$
$\langle 2,2\rangle, \langle 2,B\rangle, \langle 2,C\rangle, \langle B,B\rangle, \langle C,C\rangle\}$
$0_{T2} = 1$
$L_2 = \{\langle 2,A\rangle, \langle B,A\rangle, \langle C,A\rangle, \langle B,C\rangle\}$

Given tree A.

A **labeling function *L* for A** is a function assiging to every node in A a **label** usually a symbol or a string.
A **labeled tree** is a pair $\langle A,L\rangle$, where A is a tree, and *L* a labeling function for A.
Given tree A.
We know that A has a minimum 0.

The **leaves of** A are the maximal elements of A:
node $a \in A$ is a **leaf** of A iff for no $b \in A$: $a < b$.

A **chain in** A is a subset of A **linearly ordered** by $\leq$:
$C \subseteq A$ is a chain in A iff for all $a,b \in C$: $a \leq b$ or $b \leq a$.

A **path in** A (or **branch in** A) is a **maximal chain** in A:
chain C in A is a maximal chain iff for every chain C' in A:
if $C \subseteq C'$ then $C=C'$.

A **bar in** A is a subset of A **intersecting** every path in A.
A **cut in** A is a minimal bar in A.

Fact: every cut in A is linearly ordered by L.
Proof: suppose C is a cut in A, but not linearly ordered by L.
Then for some $a,b \in C$, either $\neg L(a,b)$ or $\neg L(b,a)$. Then, by semi-connectedness either $a \leq b$ or $b \leq a$, say, $a \leq b$.
Then $C-\{b\}$ is a bar in A. Since $C-\{b\} \subseteq C$, this contradicts the assumption that C was minimal.

Corrollary: The set of leaves in A is linearly ordered by L.
Proof: the set of leaves is a cut in A.

Let T be a labeled tree in which each leaf is labeled by a string in A*. Let the leaves be $a_1,\ldots a_n$ in left right order, and for each node a, let *l*(a) be the label of

that node.
The **yield of** T is the string $l(a_1)^\wedge \ldots ^\wedge l(a_n)$.

So, the yield of a tree is the string that you get by concatenating the strings on the leaves of the tree in left-right order.

Let *T* be a set of labeled trees of the above sort.
The **yield of** *T* is the set of strings: $\{\alpha$: for some T $\in T$: α is the yield of T$\}$

STRING REWRITE GRAMMARS

A **grammar** is a tuple $G = \langle V_N, V_T, S, P \rangle$, where:

1. V_N, the set of **non-terminal symbols,** is a finite set of symbols (category labels like, say, S, NP, VP, V).
2. V_T, the set of **terminal symbols** is a finite set of symbols (lexical items)
3. $V_N \cap V_T = \emptyset$.
   $V = V_N \cup V_T$ is called the **vocabulary**.
4. S is a designated **non**-terminal symbol, the **start symbol**.
   $S \in V_N$.
5. P is a finite set of **production rules**.
   Every production rule has the form:
   $\varphi \rightarrow \psi$
   where $\varphi, \psi \in V^*$, $\varphi \neq e$.

We read $\varphi \rightarrow \psi$ as: **rewrite string φ as string ψ.**

What this means is the following:

Let G be a grammar and Let $\alpha, \varphi, \psi \in V^*$ and let n_φ be an occurrence of string φ in α and let $\alpha[n_\psi/n_\varphi]$ be the result of replacing occurrence n_φ of φ by an occurrence n_ψ of ψ in α.
A rule $\varphi \rightarrow \psi$ allows us to rewrite string φ as string ψ **in α** if α contains an occurrence n_φ of φ, and this means: replace α by $\alpha[n_\psi/n_\varphi]$.

Example: V NP → V DET N
This rule allows us to rewrite the string: NP **V NP** PP as NP **V DET N** PP.
But it doesn't allow us to rewrite the string: John kissed NP in the garden
as: John kissed DET N in the garden.

Example: NP → DET N
This rule allows us to rewrite the string: NP **V NP** PP as NP **V DET N** PP
And it allows us to rewrite the string: John kissed NP in the garden
as: John kissed DET N in the garden.

Let G be a grammar, $\alpha, \beta, \varphi, \psi \in V^*$, $R \in P$, where $R = \varphi \rightarrow \psi$, n_φ an occurrence of φ in α.

$\alpha \Rightarrow_{G,R,n} \beta$, α **directly dominates** β **by** rule R **at** n iff $\beta = \alpha[n_\psi/n_\varphi]$
(β is the result of replacing occurrence n_φ of φ in α by occurrence n_ψ of ψ.

$\alpha \Rightarrow_{G,R} \beta$, α **directly dominates** β **by** rule R iff for some n_φ in α, $\beta = \alpha[n_\psi/n_\varphi]$

$\alpha \Rightarrow_G \beta$ iff for some rule $R \in P$: $\alpha \Rightarrow_{G,R} \beta$ (β derives from α)

Let G be a grammar, $\varphi_1, \ldots, \varphi_n \in V^*$, $R_1 \ldots R_{n-1}$ a sequence of rules from P (so rules from P may occur more than once in the sequence), $n_1 \ldots n_{n-1}$ a sequence of occurences of subformulas in $\varphi_1, \ldots, \varphi_{n-1}$ (with n_i in φ_i)

$<\varphi_1,R_1,1>\ldots.<\varphi_{n-1},R_{n-1},n\text{-}1><\varphi_n>$ is a **derivation of** φ_n **from** φ_1 **in** G iff:

for every $i<n$: $\varphi_i \Rightarrow_{G,Ri,ni} \varphi_{i+1}$

Derivation $<\varphi_1,R_1,1>\ldots.<\varphi_{n-1},R_{n-1},n\text{-}1><\varphi_n>$ is **terminated in** G iff there is **no** derivation $<\varphi_1,R_1,1>\ldots.<\varphi_{n-1},R_{n-1},n\text{-}1><\varphi_n,R_n,n><\psi>$ of some string ψ in G

Thus, a derivation of φ_n is terminated in G if no rule of G is applicable to φ_n anymore.

α is a **terminal string generated by** G iff

1. There is a **terminated** derivation in G which starts with S and ends with α, $<S,R_1,1>\ldots.<\alpha>$.
2. α is a string of **terminal symbols**, i.e. $\alpha \in V_T^*$.

The **language generated by** G, L(G), is the set of terminal strings generated by G.

Grammars G and G' are **weakly equivalent** iff L(G) = L(G')

In rule $R = \varphi \rightarrow \psi$, φ and every substring of φ occurs on the left side of R, ψ and every substring of ψ occurs on the right side of R, φ itself occurs **as** the left side of R.

Grammar G is **in reduced form** iff startsymbol S does not occur on the right side of any rule in G.

Theorem: For any grammar G, there is a grammar G' which is in reduced form such that L(G)=L(G')

Proof:
Let G be any grammar.
Add a new symbol S_0 to V_N and replace in every rule every occurrence of S by S_0.
This gives us a new set of rules P'. Call this grammar G'.
The language generated in G' from S_0 is, of course, the same language as the language generated in G from S.
Let R be a rule in the new set which contains S_0 **as** the left side.
Let $R[S/S_0]$ be the result of replacing in R S_0 **as the left side** by S, leaving the right side unchanged.
Add to P', for every such rule R, rule R[S/S0] to the grammar.
Call this grammar G_{RED}.
G_{RED} is obviously a grammar which is in reduced form (S does not occur on the right side of any rule, though S_0 may well).
And G_{RED} generates the same language as G.

This can be seen as follows.
Suppose that in G we had a terminated derivation of a terminal string α:
$<S,R_1,1><\varphi_1,R_2,2>\ldots.<\varphi_{n-1},R_{n-1},n\text{-}1><\alpha>$.
This means, that in G' we have a terminated derivation of α of the form:
$<S_0,R_1^*,1><\varphi_1,R_2^*,2>\ldots.<\varphi_{n-1},R_{n-1}^*,n\text{-}1><\alpha>$
where R_i^* is R_i if we didn't need to change it, otherwise it is the changed rule.
Since S_0 directly dominates φ_1 by R_1^*, and since S_0 is a symbol, S_0 occurs **as** the left side of R_1^*. But this means that that G_{RED} contains rule $R_1^*[S/S_0]$, and that means that in G_{RED} we have a terminated derivation of α:

<S,R_1*[S/S_0],1><φ_1,R_2*,2>….<φ_{n-1},R_{n-1}*,n-1><α>.
Thus, G_{RED} generates α.

Vice versa, we note that no rule in G_{RED} contains S on the right side. This means that there are no derivations in G_{RED} where a rule R_1*[S/S_0] occurs in any other step besides the first step. This means that for any derivation in G_{RED} of the form:
<S,R,1><φ_1,R_2,2>….<φ_{n-1},R_{n-1},n-1><α>
there is some rule R_1* in G' such that R = R_1[S/S_0] and G' contains a derivation of α from S_0: <S_0,R_{1*},1><φ_1,R_2,2>….<φ_{n-1},R_{n-1},n-1><α>.
This derivation in G' only differs from the derivation in G_{RED} in its first step.
This means that every derivation from S in G_{RED} joins, in its second step, a derivation in G'. And this means that G_{RED} does not generate any terminal string from S that G' doesn't generate as well from S_0. And this means that indeed L(G_{RED})=L(G).
This proves the theorem.

> Let G be a grammar in **reduced form**.
> **G_e** = G + S $\rightarrow$ e
> Thus, G_e is the result of adding rule S $\rightarrow$ e to G_e.

FACT: L(G_e) = L(G) $\cup$ {e}
Proof: Since S doesn't occur on the right side in any rule in G and the notion of grammar doesn't allow the leftside of any rule to be e, the only derivation that we can make in G_e that we (possibly) couldn't make before in G is the derivation: <S,S$\rightarrow$e><e>. Thus we derive e in G_e, (and note that e **does** count as a terminal string, since it **is** an empty string of terminal symbols).
I say 'possibly', because the fact also holds if L(G) already contained e.

Note the difference between: L(G)=Ø and L(G)={e}.
If G is a grammar with only rule: S $\rightarrow$ e, then L(G)={e}.
If G is a grammar with only rule S $\rightarrow$ NP VP, then L(G)=Ø.

Example

S $\rightarrow$ S S
S $\rightarrow$ a

Generated language: {a}$^+$

Reduced form:

$S_0 \rightarrow S_0\ S_0$	+	$S \rightarrow S_0\ S_0$
$S_0 \rightarrow$ a		S $\rightarrow$ a
	+	S $\rightarrow$ e

Generated language: {a}*

TYPES OF GRAMMARS

Type 0: Unrestricted rewrite grammars.

Rules of the form: $\varphi \rightarrow \psi$, where $\varphi, \psi \in V^*$, $\varphi \neq e$.

Type 1: Context sensitive grammars.

a. Rules of the form: $\varphi \rightarrow \psi$, where $\varphi, \psi \in V^*$, $\varphi \neq e$ **and** $|\varphi| \leq |\psi|$

b. If G is a context sentitive grammar in reduced form, G_e is a context sentitive grammar.

Context sensitive grammars do not allow **shortening rules**: the output of a rule must be at least as long as the input. The only exception that we make is that, if the grammar is in reduced form, we can allow $S \rightarrow e$ as the only shortening rule and still call the grammar context sensitive.

The name context sensitive comes from a different, equivalent formalization of the same class of grammars which we will mention below.

Type 2: Context free grammars:

a. Rules of the form: $A \rightarrow \psi$, where $A \in V_N$ and $\psi \in V^+$.

b. If G is a context free grammar in reduced form, G_e is a context free grammar.

In context free grammars, the rules have a non-terminal **symbol** on the left side, not a string (so there is no context), and a non-empty string on the right.
As before, we allow $S \rightarrow e$ in a context free grammar G if $G - S \rightarrow e$ is contextfree, and in reduced form. (Note, later in these notes I will liberalize the concept of context free grammar.)

Type 3: Right linear grammars:

a. Rules of the form: $A \rightarrow \alpha B$ or $A \rightarrow \alpha$, where $A,B \in V_N$ and $\alpha \in V_T^+$

b. If G is a right linear grammar in reduced form, G_e is a right linear grammar.

In right linear grammars the rules have a non-terminal **symbol** on the left side of $\rightarrow$, like in context free grammars, and on the right, a **non-empty string of terminal symbols,** or a **non-empty string of terminal symbols followed by a non-terminal symbol (i.e. one non-terminal symbol on the right side of the produced string).**
Here too, we allow $S \rightarrow e$ in a right linear grammar if the grammar is inreduced form, (Note: also this notion will be liberalized later.)

A **language** L is **type n** iff there is a type n grammar G such that L = L(G).

So a language is context free if there exists a contextf ree grammar that generates it. Note that every right linear language is context free, every context free language is context sensitive, and every context sensitive grammar is type 0. This is because a right linear grammar is a special kind of context free grammar, a context free grammar a special kind of context sensitive grammar, a context sensitive grammar a special kind of type 0 grammar.

[Note that there are languages that don't have a grammar (meaning, not even a type 0 grammar). Take alphabet {a,b}. {a,b}* is a countable set. The set of all languages in alphabet {a,b} is the powerset of {a,b}*, pow({a,b}*, and Cantor has proved that that set is uncoutable.
We can assume that the possible non-terminal symbols for grammars in alphabet {a,b} come from a fixed countable set of non-terminals. Since grammars are finite, we can represent each grammar in alphabet {a,b} as a **string** in a 'grammar alphabet': say, a string:
V_T={a}--V_N={SAB}--R={S→SS,S→ A,S→B,A→a,B→b}
(so, literally think of this as a string of symbols, i.e. { is a bracket symbol,→ is an arrow symbol …}
If the non-terminal symbols come from a fixed countable set you can prove that there are only countably many such strings, and hence there are only countably many grammars in alphabet {a,b}. This means that there are **more** languages in alphabet {a,b} than there are grammars for languages in alphabet {a,b}, and this means by necessity that **there are more languages in alphabet {a,b} that don't have a (type 0) grammar at all than languages that do**.]

We call languages that don't have a grammar **intractable languages**.

EXAMPLE
G = <V_N,V_T,S,P>
V_N = {S, A, B, C}
V_T = {a,b,c,d}

P =	1. S → ABC	(Context free rule)	
	2. A → aA	(Right linear rule)	
	3. A → a	(Right linear rule)	(Also left linear)
	4. B → Bb	(Left linear rule)	
	5. B → b	(Right linear rule)	
	6. Bb ⟶ bB	(Context sensitive rule)	
	7. BC → Bcc	(Context sensitive rule)	
	8. ab → d	(Type 0 rule)	
	9. d → ba	(Context sensitive rule)	

This is a Type 0 grammar.

Derivations:
BCA ,7
BccA ,4
BbccA ,2
BbccaA ,3
Bbccaa ,5
bbccaa

This is a terminated derivation of bbccaa from BCA

S ,1
ABC ,3
aBC ,7
aBcc ,5
abcc ,7
dcc ,8
bacc

This is a terminated derivation of bacc from S, hence bacc is in the generated language.

S ,1
ABC ,3
aBC ,7
aBcc ,5
abcc

This is a derivation of abcc from S, but not a terminated one. This derivation does not guarantee that abcc is in the generated language (in fact, it is not).

S ,1
ABC ,3
aBC ,5
abC

This is a terminted derivation of abC from S. Since abC is not a string of terminals, abC is not in the generated language, even though the derivation is from S and terminated.

The generated language is: $b^n a^m$ cc $(n,m \geq 1)$
This expresses that each string consists of a string of b's (minimally one), followed by a string of a's (minimally one), where the number of b's and a's are unrelated, followed by two c's.
We know that this language is type 0. But, in fact, the language is also generated by the following grammar:

G = <{S,A,B,C},{a,b,c},P> where
P = 1. $S \rightarrow bB$
2. $B \rightarrow bB$
3. $B \rightarrow aA$
4. $A \rightarrow aA$
5. $A \rightarrow cC$
6. $C \rightarrow c$

Since this grammar is right linear, it follows that $b^n a^m$ cc $(n,m \geq 1)$ is a right linear language.

DERIVATIONS AND PARSE TREES

Type 1 Context sensitive
a. Rules of the form $\varphi \rightarrow \psi$,
where $\varphi, \psi \in V^*$, $\varphi \neq e$ and $|\varphi| \leq |\psi|$
b. If G is in reduced form, G_e is type 1

Type 1* Context sensitive in Normal Form
a. Rules of the form $\alpha A \beta \rightarrow \alpha \varphi \beta$,
where $A \in V_N$, $\alpha, \beta, \varphi \in V^*$ $\varphi \neq e$
b. If G is in reduced form, G_e is type 1
(Note: such rules are type 1, since they are not shortening)

Type 1**
a. Rules of the form $A \rightarrow \varphi / \alpha \ldots \beta$
where $A \in V_N$, $\alpha, \beta, \varphi \in V^*$ $\varphi \neq e$
Rewrite A as φ in (local) context $\alpha \ldots \beta$
b. If G is in reduced form, G_e is type 1

Example:
$BC \rightarrow BcC$
This rule is in normal form, but not uniquely:
$B \rightarrow Bc/ \ldots C$
$C \rightarrow cC/B \ldots$

Theorem: For every context sensitive grammar there is an equivalent context sensitive grammar in Normal Form of type 1*. [proof omitted]

Linguistically oriented books often give the type 1** definition of context sensitive grammars, for historic reasons: rules in phonology used to have this format. However, the format was by and large only used in phonology to summarize finite paradigms, and it turns out to be a really rotten format to write grammars in for the languages that we are actually going to show to be context sensitive, while writing grammars in the no shortening format is easy. Since the formats are proved to be equivalent, I will not ask you to give grammars in the second format.

I will introduce one more new concept:

> A **context desambiguated rule** is a pair $<R,<\alpha,\beta>>$, where R is a context senstive rule in normal form: $\alpha A\beta \rightarrow \alpha\psi\beta$.
>
> A **context desambiguated grammar** is a structure $G = <V_N,V_T,S,R>$, with V_N,V_T,S, as usual, and R a finite set of context desambiguated rules.

The notion of derivation stays exactly the same.
We now specify in all our context sentive rules what the context is. But, apart from that, the notion of derivation doesn't **do** anything with that context specification. This means that, if we leave out from our context desambiguated grammar, all the context specifications, we get an equivalent context sensitive grammar in normal form.

Hence, the format of context desambiguated grammars puts no restriction on the generated languages: the class of all languages generated by context sensitive grammars is exactly the class of languages generated by context desambiguated grammars.

The difference is as follows. We could, in a context sensitive grammar, have a rule of the form: R: $BC \rightarrow BcC$. This rule is in fact in normal form, except that it isn't **uniquely** in normal form. We can interpret this rule as:

> $\alpha B\beta \rightarrow \alpha Bc\beta$, where α=e and β=C: rewrite B as Bc, when followed by C.

but also as:

> $\alpha C\beta \rightarrow \alpha bC\beta$, where α=B and β=e: rewrite C as cC, when preceded by B.

The only thing that the context desambiguated format does is specify which of these interpretations is meant:
In the context desambiguated format we have two different rules:

> R_1: $<BC \rightarrow BcC, <e,C>$: rewrite B to Bc before C.
> R_2: $<BC \rightarrow BcC, <B,e>$: rewrite C to cB after B.

And the grammar may contain R_1, or R_2, or both. As said, this makes no difference in generative capacity, since at each step of the derivation, the effect of applying R_1 or R_2 to an input string is exactly the same as applying R to that input string: BC gets rewritten as BcC.

But the context desambiguated format is useful when we want to assign parse-trees to derivations.

Let G be a context-desambiguated grammar, and let $D = <B,R_1,1>...<\varphi_n>$ be a derivation in G of φ from B, with $B \in V_N$ of n steps.

> The **parse-tree determined by** D is the tree T_D with topnode labelel B, yield φ_n, non-leaf nodes labeled by non-terminal symbols in V_N constructed in the following way:
> 1. We associate with the first step of D a tree T_1 , which is a tree with a single node labeled B.

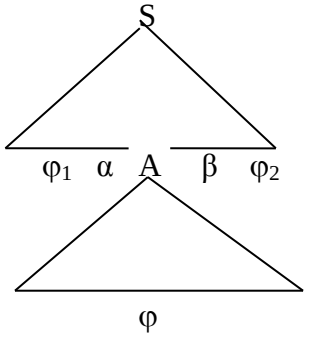
2. Let $<\varphi_1\alpha A\beta\varphi_2,<R_i,<\alpha,\beta>>,i>$ be step i in D,
let T_i be the tree constructed for step i,
and let $<\varphi_1\alpha\varphi\beta\varphi_2,R_{i+1},i+1>$ be step i+1 in D.
$\varphi_1\alpha A\beta\varphi_2$ is the yield of T_i.
In this, $\alpha A\beta$ is the occurrence i, so each symbol in $\varphi_1\alpha A\beta\varphi_2$ unambiguously labels a leafnode in T_i (this is the reason for mentioning the occurrence i in the derivation). Take the leafnode in T_i that the mentioned occurrence of A labels.
Call this node n.
Tree T_{i+1} is the tree which you get from T_i by attaching to node n in T_i **as daughter nodes**, nodes labeled by the symbols in φ, in left-right order, one symbol per node. Clearly, the yield of T_{i+1} is $\varphi_1\alpha\varphi\beta\varphi_2$.
3. The parse-tree determined by D is tree T_n.

Let G be a context-desambiguated grammar.
A **parse tree of** G is a tree which is the parse tree of some derivation of G.

It is easy to see that the following fact holds:

FACT: In a context desambiguated grammar each derivation determines a **unique** parse tree.

This is, because in a context desambiguated grammar, the description of input step i: $<\varphi_1\alpha A\beta\varphi_2,<R_i,<\alpha,\beta>>,i>$ determines a unique node in T_i for A. This is not necessarily the case for context sensitive grammars that are not in the context desambiguated format. For example:
For a step like: $<\varphi_1 BC\varphi_2$, $BC\rightarrow BcC,i>$, the algorithm for constructing the parse tree wouldn't know whether to attach B and c as daughters to B or c and C as daughters to C.
Desambiguation specifies precisely that information:
In a step: $<\varphi_1 BC\varphi_2$, $<BC\rightarrow BcC,<B,e>>,i>$, you are instructed to attach c and C as daughters to C.
In a step: $<\varphi_1 BC\varphi_2$, $<BC\rightarrow BcC,<e,C>>,i?$, you are instructed to attach B and c as daughters to B.



Since contex free grammars **are** context desambiguated (every rule has context <e,e>), we get:

CORROLARY: In a context free grammar each derivation determines a **unique** parse tree.

Let G be a context desambiguated grammar.
A **constituent structure tree generated by** G or **I-tree generated by** G is a parse tree T for G with topnode S and yield(T) $\in$ L(G), corresponding to a terminated derivation in G of a terminal string from S

The **tree set of** G, T(G), is the set of all I- trees generated by G.

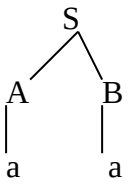
Let G and G' be context-desambiguated grammars.
G and G' are **strongly equivalent** iff T(G)=T(G').

It is easy to see that if G and G' are strongly equivalent, G and G' are weakly equivalent. The inverse does not hold, though. Different grammars may generate the same strings with different structures.
Also, **the same** grammar may generate a string with different structures:

Let G be a context desambiguated grammar and let $\varphi \in$ L(G).
φ is **syntactically ambiguous** in G iff φ is the yield of two I-trees generated by G.

G is **syntactically ambiguous** if some $\varphi \in$ L(G) is syntactically ambiguous in G.

There is an advantage to characterizing syntactic ambiguity in terms of I- trees rather than directly in terms of derivations. I- trees do not reflect the **order** in which the derivations build them.
If the grammar generates string aa with structure:



we want to call aa unambiguous, even though the grammar may well have different derivations for this structure:
$S \Rightarrow AB \Rightarrow aB \Rightarrow aa$
$S \Rightarrow AB \Rightarrow Ab \Rightarrow aa$
This is derivational ambiguity which is non-structural. With the notion of constituent structure trees we can express the difference.