# *An introduction to multiple context free grammars for linguists*

*Alexander Clark*

*September 20, 2014*

This is a gentle introduction to Multiple Context Free Grammars (MCFGS), intended for linguists who are familiar with context free grammars and movement based analyses of displaced constituents, but unfamiliar with Minimalist Grammars or other mildly context-sensitive formalisms.[1]

[1] I originally wrote this for LOT 2012. This is a lightly edited version with a few corrections.

## *Introduction*

So to start, a very small bit of historical background – not by any means a proper survey. We are all familiar with the Chomsky hierarchy [2]. This defined a collection of different formalisms that have various types of power. This was an incredible intellectual accomplishment, but Chomsky (unsurprisingly) didn't get it absolutely right first time. Before Chomsky's pioneering work, there were three very natural classes of languages – the class of finite languages, the class of regular languages, and the class of computable languages.

[2] N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3): 113–124, 1956

Chomsky identified the class of context free grammars (CFGs), and he recognised (correctly, but on the basis of incorrect arguments) that this was insufficient for describing natural languages.

We blur here some technical distinctions between recursive, primitive recursive and recursively enumerable languages that are not relevant to the discussion.

In the context of linguistics, it is clear that Chomsky's initial attempt to formalise a richer model than context free grammars ended up with a class that is much too powerful to be useful: the context sensitive grammars are equivalent just to Turing machines with a certain bound and have a completely intractable parsing problem. More formally, determining whether a string is generated by a context sensitive grammar is decidable, in that it can be determined by a computer in a well defined way, but it would require far too many computational steps to do in general. If we are interested in cognitively real representations, then given that the human brain has a limited computational capacity, we can safely assume that the representations that are used can be efficiently processed. When it finally became clear that natural languages were not weakly context free [3] interest thus shifted to classes that are slightly more powerful than context free grammars while still having a tractable parsing problem – these are the *mildly context-sensitive formalisms* [4]. There has been a great deal of work on this recently, which we won't try to summarise completely – here we describe what we think of as the simplest and most natural extension of CFGs, the class of Multiple Context Free Grammars introduced by Seki et al. [5]. These turned out to be equiv-

[3] S. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343, 1985

[4] K. Vijay-Shanker and David J. Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27(6):511–546, 1994

[5] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):229, 1991

alent to Minimalist Grammars [6] under certain assumptions. In this note we will try to give a non technical explanation of MCFGs in a way that is accessible to linguists and in particular to explain how these can give a natural treatment of the types of syntactic phenomena that in mainstream generative grammar have been treated using movement.

[6] E. Stabler. Derivational minimalism. In C. Retoré, editor, *Logical aspects of computational linguistics (LACL 1996)*, pages 68–95. Springer, 1997

One important property of MCFGs that they share with CFGs, is that they define a natural notion of hierarchical structure through their derivation trees. Chomsky and many others often emphasize the importance of considering structural descriptions rather than just surface strings: of *strong generative capacity* rather than just the *weak generative capacity* considered in formal language theory. MCFGs, just as with CFGs, have derivational structures that naturally give a hierarchical decomposition of the surface string into constituents – in the case of MCFGs as we shall see, these constituents can be discontinuous or contain displaced elements.

However MCFGs also inherit some of the weaknesses of CFGs – in particular, they lack an appropriate notion of a feature. Just as with CFGs, modeling something as simple as subject verb agreement in English, requires an expansion of the nonterminal symbols: replacing symbols like $NP$ with sets of more refined symbols like $NP_{3pers-sing}$ and $NP_{1pers-plur}$ that contain person and number features. In MCFGs, the nonterminals are again atomic symbols, and thus a linguistically adequate system must augment the symbols using some appropriate feature calculus. We don't consider this interesting and important problem here, but focus on the more primitive issue of the underlying derivation operations, rather than on how these operations are controlled by sets of features.

## *Context free grammars*

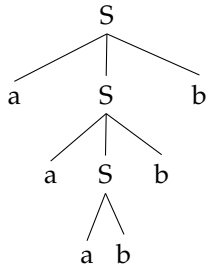
We will start by looking at context free grammars and reconceiving them as being a bottom up rather than a top down derivation, and we will switch to a new notation. The switch from a top-down derivation to a bottom-up is in line with current thinking in the Minimalist Program which hypothesizes a primitive operation called MERGE which combines two objects to form another.

Consider a simple CFG that has one nonterminal $S$, and productions $S \to ab$ and $S \to aSb$. This generates the language $L$ which consists of $ab, aabb, aaabbb, \ldots$ indeed all strings consisting of a nonzero sequence of $a$s followed by an equal number of $b$s. Here $S$ is a nonterminal and $a$ and $b$ are terminal symbols. If we consider the string $aaabbb$, we would say there is a valid derivation which goes something like this:

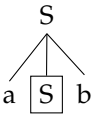
In maths we would say $L = \{a^n b^n | n > 0\}$.

1. We start with the symbol $S$.
2. We rewrite this using the production $S \to aSb$ to get $aSb$.
3. We rewrite the symbol $S$ in the middle of $aSb$ using the production $S \to aSb$ to get $aaSbb$.
4. We rewrite this using the production $S \to ab$ to get $aaabbb$. We don't have any nonterminal symbols left so we stop there.

It is much better to think of this as a tree.



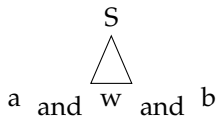

We can view the CF production $S \to aSb$ as an instruction to build the tree starting from the top and going down: that is to say if we have a partial tree like this:



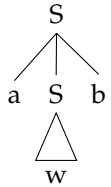

We can view the productions as saying that we can replace the boxed S node in the tree with a subtree either of the same type or of the type:


An alternative way is to view it as a bottom-up construction. We can think of the production $S \to ab$ as saying, not that we can rewrite $S$ as $ab$ but rather that we can take an $a$ and a $b$ and we can stick them together to form an $S$. That is to say the production $S \to aSb$ says that we can take the three chunks:

The switch to bottom up derivations is also a part of the Minimalist Program.




and combine them to form a new unit like this



This suggests a change in notation: from $S \rightarrow aSb$ to $S \leftarrow aSb$. Let us do something a little less trivial.

We have the standard notation $N \stackrel{*}{\Rightarrow} w$ which means that $N$ can be rewritten through zero or more derivation steps into the string $w$. Informally this means that $w$ is an $N$ – $S \stackrel{*}{\Rightarrow} aabb$ means that $aabb$ is an $S$. We can write this more directly just as a predicate:

$$S(aabb). \tag{1}$$

This says directly that $aabb$ satisfies the predicate $S$. In an English example, we might say NP(the cat), or VP(died).

A production rule of the form $N \rightarrow PQ$ then means, viewed bottom up, that if we have some string $u$ which is a $P$ and another string $v$ that is a $Q$ we can concatenate them and the result will be an $N$. The result of the concatenation is the string $uv$: so we can write this rule as an implication:

If $u$ is a $P$ and $v$ is a $Q$, then $uv$ is an $N$.

Writing this using the predicate notation this just says: $P(u)$ and $Q(v)$ imples $N(uv)$ We will write this therefore as the following production:

$$N(uv) \leftarrow P(u)Q(v) \tag{2}$$

Here $N, P, Q$ are nonterminals and $u$ and $v$ are variables. We will use letters from the end of the alphabet for variables in what follows. We don't have any terminal symbols in this rule.

This notation is due to Makoto Kanazawa and is based on a logic programming notation. Originally in Prolog this would be written with ":=" instead of $\leftarrow$. These are sometimes called Horn clauses.

Productions like $S \rightarrow ab$ turn into rules like $S(ab) \leftarrow$ where there is nothing on the right hand side, because there are no nonterminals on the righthand side of the rule. So we will just write these as $S(ab)$, and suppress the arrow. We can view this as a starightforward asertion that a string $ab$ is in the set of strings derived from $S$.

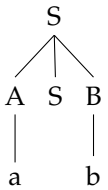
Let us go back to the trivial example before ( $S \rightarrow aSb$). If $w$ is an $S$, then $awb$ is also an $S$: we write this as

$$S(awb) \leftarrow S(w) \tag{3}$$

Note that the terminal symbols $a$ and $b$ end up here on the left hand side of the rule. This is because we know that an $a$ is an $a$!

We could define predicates like $A$ which is true only of $a$, and then have the rule $S(uwv) \leftarrow A(u), S(w), B(v)$, together with the rules with an empty righthand side $A(a)$ and $B(b)$.

Here $A$, and $B$ would be like preterminals in a CFG, or lexical categories in some sense:




Let us consider a point which is kind of trivial here but becomes nontrivial later. Compare the two following rules:

$$N(uv) \leftarrow P(u)Q(v) \tag{4}$$

$$N(uv) \leftarrow Q(v)P(u) \tag{5}$$

If you think about this for a moment, they say exactly the same thing. Two context free rules $N \rightarrow PQ$ and $N \rightarrow QP$ are different because we express the two different concatenations $uv$ versus $vu$ by the order of the nonterminals on the right hand side of the rule. In the case of this new format, we express the concatenation explicitly using variables. This means we have a little bit of spurious ambiguity in the rules. We can stipulate that we only want the first one – for example by requiring that the order of the variables on the left hand side must match the order of the variables in the right hand side.

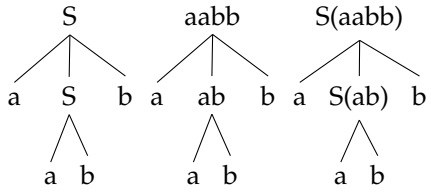
Think a little more here – if we have two string $u$ and $v$ there are basically two ways to stick them together: $uv$ and $vu$. We can express this easily using the standard CFG rule, just by using the two orders $PQ$ and $QP$. But are there only two ways to stick them together? What about $uuv$? Or $uuuu$? Or $uvuvvuuu$? Actually there are, once we broaden our horizons, a whole load of different ways of combining $u$ and $v$ beyond the two trivial ones $uv$ and $vu$. But somehow the first two ways are more natural than the others because they satisfy two conditions: first, we don't copy any of them, and secondly we don't discard any of them. That is to say the total length of $uv$ and $vu$ is going to be just the sum of the lengths of $u$ and $v$ no matter how long $u$ and $v$ are. Each variable on the right hand side of the rule occurs exactly once on the left handside of the rule – and there aren't any other variables. $uuuv$ is bad because we copy $u$ several times, and $uu$ is bad because we discard $v$ (and copy $u$). We will say that concatenation rules that satisfy these two conditions are *linear*.

So the rule $N(uv) \leftarrow P(u)Q(v)$ is linear, but $N(uvu) \leftarrow P(u)Q(v)$ is not.

Summarising, the grammar for $a^n b^n$ above just has two productions that we write as $S(ab)$ and $S(awb) \leftarrow S(w)$.

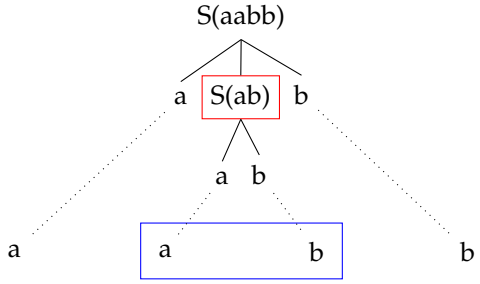
When we have a derivation tree for a string like *aabb* we can write this with each label of the tree being a nonterminal as we did earlier — this is the standard way — or we can write it where each node is

labeled with the string that that subtree derives, or we can label them with both.




The last of these three trees has the most information, but if we have long strings, then the last two types of tree will get long and unwieldy.

Finally, when we look at the last two trees, we can see that we have a string attached to each node of the tree. At the leaves of the tree we just have individual letters, but at the nonleaf nodes we have sequences that may be longer. Each of these sequences corresponds to an identifiable subsequence of the surface string. That is to say, when we consider the node labeled $S(ab)$ in the final tree, the $ab$ is exactly the $ab$ in the middle of the string.




This diagram illustrates this obvious point: we have drawn the surface string $aabb$ below the tree. Each leaf node in the tree is linked by a dotted line to the corresponding symbol in the surface string. We have boxed in red the nonleaf node labeled $S(ab)$) and boxed in blue the corresponding subsequence of the surface string $ab$. In a CFG we can always do it, and the subsequence will always be a contiguous subsequence – i.e. one without gaps. This is very easy to understand if you are familiar with CFGs: the point here is to introduce the diagram/notation in a familiar context so that when we reuse it for MCFGs it is easy to understand there too. Thus the derivation tree of the CFG defines a hierarchical decomposition of the surface string into contiguous subsequences that do not overlap, but may include one another. The root of the tree will always correspond to the whole string, and the leaves will always correspond to individual symbols in the string.

## *Multiple Context Free Grammars*

There are many linguistic phenomena where it is impossible or difficult to find an adequate analysis using just the machinery of context free grammars, and simple trees of this type, and as a result, almost immediately after CFGs were invented by Chomsky, he augmented them with some additional machinery to handle it. The particular augmentations he considered – transformations – have gone through various iterations over the years. Here we will take a different direction that is more computationally tractable and discuss the relationship to the movement based approach. We will discuss some examples, like Swiss German cross serial dependencies in a bit of detail below, but even a very simple example like a relative clause in English causes some problems.

(6) This is the book that I read.

(7) This is the book that I told you to read.

The problem is that it is difficult to find a construction that supports a compositional semantics for this sentence – "book" must be simultaneously a head of the NP (or something similar in a DP analysis), and the object in some underlying sense of the verb "read". So in a sense it needs to be in two places at once.

Traditionally we have some idea of movement where the noun "book" starts off in some underlying position as an object of "read" and then moves to its final position as shown in this figure.

Computationally it turned out that this movement operation, in its unconstrained form, violated some fundamental properties of efficient computation [7]. There is an alternative way of modeling this data that turns out to be exactly equivalent to a restricted form of the movement analysis, which also is computationally efficient and thus cognitively plausible in the way that the unconstrained movement approaches are not.

[7] P .S. Peters and R. W. Ritchie. On the generative power of transformational grammars. *Information Sciences*, 6:49–83, 1973

We will now define the extension from CFGs to MCFGs, and then we will discuss the linguistic motivation in some depth. The key element is to extend the predicates that we have in our CFG, so that instead of applying merely to strings, they apply to pairs of strings. These pairs of strings have a number of different linguistic interpretations. In a CFG which generates individual strings, we take these strings to be constituents. In an MCFG, one interpretation of the pair is as a discontinuous constituent — that is to say, we have something that we want to be a constituent, but which has some material inserted in the middle that we want not to be part of that constituent. We can consider this as a constituent with a gap in it, and the pair

In full generality, MCFGs form a hierarchy where nonterminals can derive not just single strings or pairs of strings, but tuples of arbitrary arity. Here for presentational purposes we will restrict ourselves to the simpler case of pairs of strings, which may in any event be sufficient for natural language description.

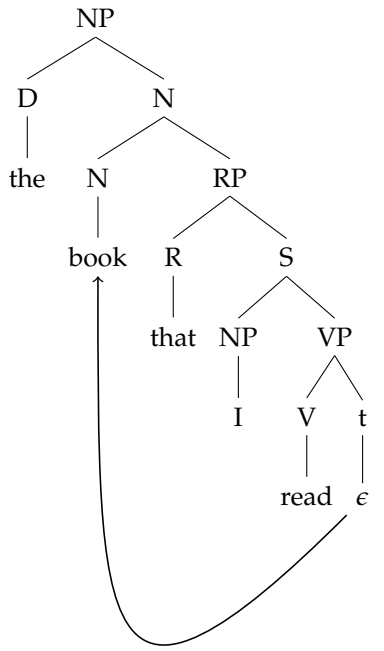



Figure 1: Tree diagram with movement – the labels on the nodes and the specific analysis are not meant to be taken seriously. There are a number of competing analyses for this construction.

of strings will correspond to the two parts of this: the first element of the pair will correspond to the segment of the constituent to the left of the gap (before the gap) and the second element of the pair will correspond to the segment after the gap. This is not the only interpretation: we might also have some construction that would be modeled as movement – in that case we might have one part of the pair representing the moving constituent, and the other part representing the constituent that it is moving out of.

So in our toy examples above we had a nonterminal $N$ and we would write $N(w)$ to mean that $w$ is an $N$. We will now consider a *two-place* predicate on strings, that applies not just to one string, but to two strings. So we will have some predicates, say $M$ which are two place predicates and we will write $M(u, v)$ to mean that the ordered pair of strings $u, v$ satisfies the predicate $M$. Rather than having a nonterminal generating a single string $S \stackrel{*}{\Rightarrow} aabb$, we will have a nonterminal that might rewrite a pair of strings $M \stackrel{*}{\Rightarrow} (aa, bb)$. Since we want to define a formal language, which is a set of strings, we will always want to have some "normal" predicates which just generate strings which will include $S$. Terminologically, we will say that all

of these nonterminals have a *dimension* which is one if it generates a single string and two if it generates a pair of strings.

Unsurprisingly we can also define MCFGs which have symbols of dimension greater than 2.

To avoid confusion in this early stage, we will nonstandardly put a subscript on all of our nonterminal symbols/predicates to mark which ones generate strings and which ones generate pairs of strings: that is to say we will subscript each symbol with the dimension of that symbol. Thus $S$, the start symbol, will always have dimension 1, and thus will be written as $S_1$. If we have a symbol $N$ that derives pairs of strings, we will write it as $N_2$. So we might write $S_1 \stackrel{*}{\Rightarrow} u$ and $N_2 \stackrel{*}{\Rightarrow} (u, v)$. In a CFG of course, all of the symbols have dimension 1.

A CFG indeed is just the special case of an MCFG of dimension 1.

Let us consider a grammar that has a nonterminal symbol of dimension two, say $N_2$, in addition to a nonterminal $S_1$ of dimension 1, and let's consider what the rules that use this symbol might look like. Here are some examples:

$$S_1(uv) \leftarrow N_2(u, v) \tag{8}$$

This rule simply says that if $N$ derives the pair of strings $u, v$ then $S$ derives the string $uv$. Note the difference – $u, v$ is an ordered pair of two strings, whereas $uv$ is the single string formed by concatenating them. Thus $N$ has a bit more structure – it knows not just the total string $uv$, but also it knows where the gap is in the middle.

$$N_2(au, bv) \leftarrow N_2(u, v) \tag{9}$$

This one is more interesting. Informally this says that if $N$ derives $(u, v)$ then it also derives the pair $(au, bv)$. This rule is clearly doing something more powerful than a CFG – it is working on two different chunks of the string at the same time. At the same time, in the same derivational step, we are adding an $a$ in one place and a $b$ in another place.

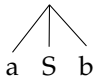
$$N_2(a, b). \tag{10}$$

Finally we have a trivial rule – this just asserts that $(a, b)$ is generated by $N_2$. This is just an analog of the way that $S \rightarrow ab$ becomes $S(ab)$, but in this case we have an ordered pair of strings rather than a single string.

An MCFG grammar consists, just like a CFG grammar, of a collection of nonterminals and some productions. Let's look at the grammar that has just these three rules. What language does it define? Just as with a CFG, we consider the language defined by an MCFG to be the set of strings generated by the symbol $S_1$: since this has dimension 1 by definition, we know that it generates a set of strings and not a set of pairs of strings. $N_2$ on the other hand generates a set of pairs of strings. Let us think what this set is. To start off with

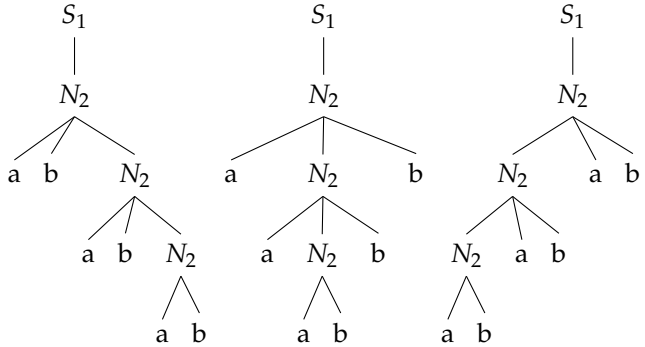
we know that it contains $a, b$, by the third production. If it contains $a, b$ then it also contains $aa, bb$ by the second rule. More slowly: the second rule says that if $N_2$ derives $u, v$ then it derives $au, bv$. Since $N_2$ derives $a, b$, setting $u = a$ and $v = b$, this means that $au = aa$ and $bv = bb$ so $N_2$ derives $aa, bb$. Repeating this process, we can see that $N_2$ derives $aaa, bbb$ and $aaaa, bbbb$ and indeed all pairs of the form $a^n, b^n$ where $n$ is a finite positive number. So what does $S_1$ derive? Well, it just concatenates the two elements generated by $N$ so this means it generates the language $ab, aabb, aaabbb, \ldots$ which is just the same as the CFG we defined before.

This might seem a letdown, but let's look a little more closely at the trees that are generated by this grammar. In particular let's look at the derivation tree for $aaabbb$.

When we try to write down a tree we immediately have a problem: what do the local trees look like? In a CFG it is easy because the order of the symbols on the right hand side of the productions tell you what order to write the symbols in. For example we have a rule $S \to aSb$, and so we write a tree like:  where the children of



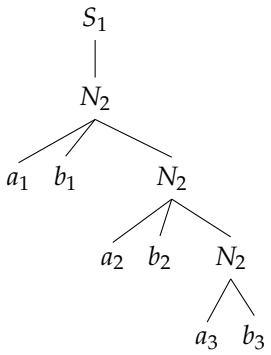
the tree occur in exactly the order of the symbols on the right hand side – $aSb$. But as we saw earlier, we don't have such a rigid notion of order, as the order is moved onto the specific function that we put on the left hand side of the rule. So we could write it in a number of ways:





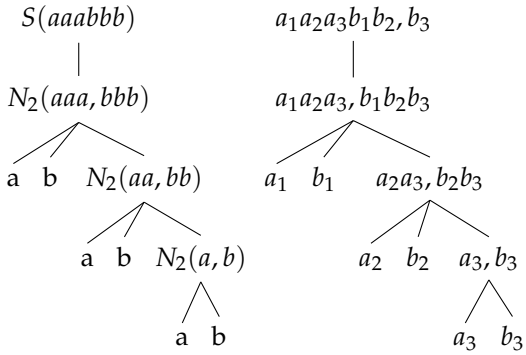
This difference doesn't matter. What matters crucially is this: while we have defined exactly the same language as the CFG earlier, the structures we have defined are completely different – in each local tree we have an $a$ and a $b$ just as with the CFG, but think about *which* pairs of $a$s and $b$s are derived simultaneously. To see this easily

let's just label the tokens as follows:

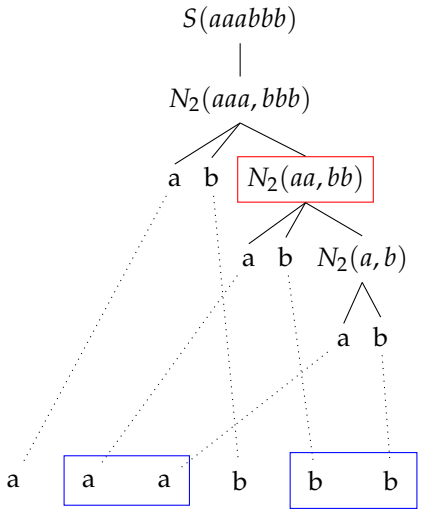



Working bottom up in this tree we see that the bottommost $N_2$ derives the pair $a_3, b_3$, the next one up derives $a_2a_3, b_2, b_3$ and the top one derives $a_1a_2a_3, b_1, b_2, b_3$, and thus $S$ derives $a_1a_2a_3b_1, b_2, b_3$. Thus we have cross serial dependencies in this case as opposed to the hierarchical dependencies.

We can write this into the tree like this as what is sometimes called a value tree.




Let us now draw the links from the nodes in the derivation trees to the symbols in the surface string as before.




Note that the red boxed symbol derives a discontinuous constituent, which we have boxed in blue. As before, since we have

restricted the rules in some way, we can identify for each node in the derivation tree a subsequence (which might be discontinuous) of symbols in the surface string.

Clearly, this gives us some additional descriptive power. Let's look at a larger class of productions now. Suppose we have three nonterminals of dimension 2: $N_2, P_2$ and $Q_2$, and consider productions that from $N$ from $P$ and $Q$.

A simple example looks like this:

$$N_2(ux, vy) \leftarrow P_2(u, v), Q_2(x, y) \tag{11}$$

Here we have four variables: $u, v, x, y$. $u$ and $v$ correspond to the the two parts of the pair of strings derived from the nonterminal symbol of arity 2, $P_2$. $x$ and $y$ correspond to the two parts derived by the symbol $Q_2$. We can see now why we need this switch in notation: if we have two single strings, then there are two ways of concatenating them, and so we can comfortably represent this with the order of the symbols on the right hand side of the production. If we have two pairs of strings, say $u, v$ and $p, q$ there are many more ways of concatenating them together to form a single pair: $up, vq$, $vq, up$ $uq, vp$, $upq, v$ and so on. We also have a load of other ways like $uuuu, pvqu$ and $uuuu, vvv$ that we might want to rule out for the same reasons that we ruled out the rules in CFGs above. So we will just consider rules which are linear in the sense we defined above – namely that each variable on the right hand side occurs exactly once on the left hand side. We also want to consider another condition which rules out productions like:

$$N_2(vu, xy) \leftarrow P_2(u, v), Q_2(x, y) \tag{12}$$

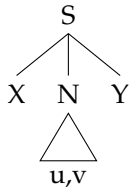
Here we have $vu$ on the left hand side, and $P_2(u, v)$ on the right hand side. This has the property that the order of the variables $u, v$ in the term $vu$ is different from the order in $P_2(u, v)$. This is called a *permuting* rule and some of the time we want to rule it out. If a rule is not permuting we say it is *non-permuting*. So for example:

$$N_2(xuv, y) \leftarrow P_2(u, v), Q_2(x, y) \tag{13}$$

This rule is non-permuting – $u, v$ occur in the right order and so too do $x, y$. We do have $x$ occurring before $u$, but that is ok – the permuting condition only holds for variables that are introduced by the same nonterminal – so $u$ has to occur before $v$ and $x$ has to occur bfore $y$. The effect of this constraint on the rules, and the linearity constraint is that it means that in a derivation, if we have a nonterminal deriving a pair of strings $u, v$ we can work out exactly which bit

of the final string corresponds to the $u$ and the $v$, and moreover we will know that the string $u$ occurs before the string $v$.

That is to say if we have an MCFG derivation tree that looks like this.




We know that the final string will look like $lumvr$ – that is to say we know that there will be a substring $u$ and a substring $v$ occuring in that order, together with some other material before and after it. This is the crucial point from a linguistic point of view – $u, v$ forms a discontinuous constituent. We can simultaneously construct the $u$ and the $v$ even if they are far apart from each other. From a semantic viewpoint this gives us a domain of locality which is not local with respect to the surface order of the string – this means that for semantic intepretation, a word can be "local" at two different points in the string at the same time.

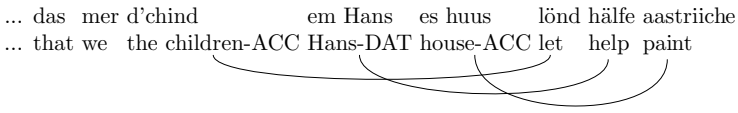
Crucially for the computational complexity of this, the formalism maintains the context free property in a more abstract level. That is to say, the validity of a step in the derivation does not depend on the context that the symbol occurs in, only on the symbol that is being processed. Thinking of it in terms of trees, this means that if we have a valid derivation with a subtree headed with some symbol, whether it is of dimension one or two, we can freely swap that subtree for any other subtree that has the same symbol as its root, and the result will be also be a valid derivation.

### *Noncontext-free languages*

The language we defined before was a context-free language and could be defined by a context free grammar, so in spite of the fact that the structures we got were not those that could be defined by a context free grammar, it wasn't really exploiting the power of the formalism. Let's look at a classic example, indeed *the* classic example, of a linguistic construction that requires more than context free power: the famous case of cross-serial dependencies in Swiss German [8].

The figure above shows the dependencies in the standard example. The crucial point is that we have a sequence of noun phrases that are marked for either accusative or dative case, and a sequence of verb phrases that require arguments that are marked for either accusative or dative case, and, finally, the dependencies overlap in the way shown in the diagram. To get this right requires the sort of

[8] S. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343, 1985




'... that we let the children help Hans paint the house'

Figure 2: Swiss German cross serial dependencies, taken from Shieber (1985). This would be preceded by something like "Jan säit" (Jan says).

MCFG apparatus that we have developed.

Let's abstract this a little bit and consider a formal language for this non context free fragment of Swiss German. We consider that we have the following words or word types: $N_a, N_d$ which are respectively accusative and dative noun phrases, $V_a, V_d$ which are verb phrases that require accusative and dative noun phrases respectively, and finally $C$ which is a complementizer which appears at the beginning of the clause. Thus the "language" we are looking at consists of sequences like

(14) $CN_aV_a$

(15) $CN_dV_d$

(16) $CN_aN_aN_dV_aV_aV_d$

but crucially does not contain examples where the sequence of accusative/dative markings on the noun sequence is different from the sequence of requirements on the verbs. So it does not contain $CN_dV_a$, because the verb requires an accusative and it only has a dative, nor does it include $CN_aN_dV_dV_a$, because though there are the right number of accusative and dative arguments (one each) they are in the wrong order – the reverse order.

Actually this is incorrect: according to Shieber the nested orders are acceptable as well. We neglect this for ease of exposition.

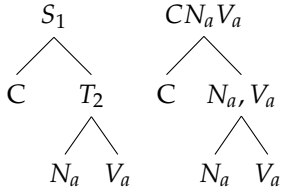
We can write down a simple grammar with just two nonterminals: we will have one nonterminal which corresponds to the whole clause $S_1$, amd one nonterminal that sticks together the nouns and the verbs which we will call $T_2$. No linguistic interpretation is intended for these two labels — they are just arbitrary symbols. As the subscripts indicate, the $S_1$ is of dimension 1 and just derives whole strings, and the $T_2$ has dimension 2, and derives pairs of strings. The first of the pair will be a sequence of $N$s and the second of the pair will be a matching sequence of $V$s.

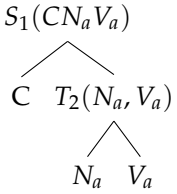
We will have the following rules:

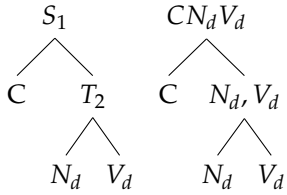
$$S_1(Cuv) \leftarrow T_2(u, v) \tag{17}$$
$$T_2(N_a, V_a) \tag{18}$$
$$T_2(N_d, V_d) \tag{19}$$
$$T_2(N_a u, V_a v) \leftarrow T_2(u, v) \tag{20}$$
$$T_2(N_d u, V_d v) \leftarrow T_2(u, v) \tag{21}$$

The first rule has a right hand side which directly introduces the terminal symbol $C$ and concatenates the two separate parts of the $T_2$. The next two rules just introduce the simplest matching of noun and verb, and the final two give the recursive rules that allow a potentially unbounded sequence of nouns and verbs to occur. Note that in the final two rules we add the nouns and verbs to the left of the strings $u$ and $v$. This is important because, as you can see by looking at the original Swiss german example, the topmost nound and verb occur on the left of the string.

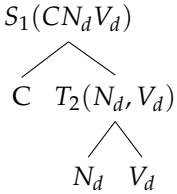
Let us look at how this grammar will generate the right strings and structures for these examples. We will put some derivation trees for the examples above, together with the matching value trees on the right. We will start with the two trivial ones which aren't very revealing. First we have the example $CN_aV_a$:



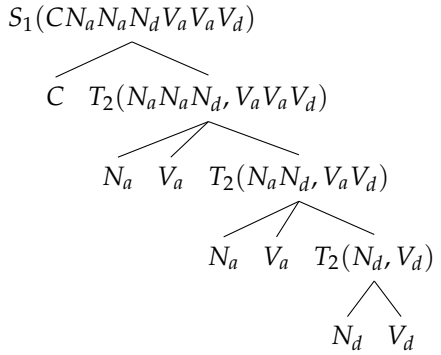



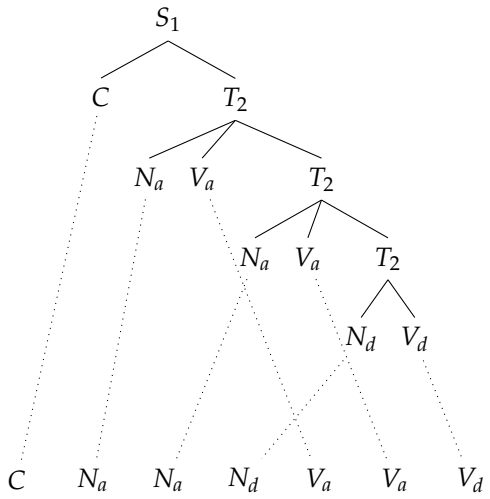

Now we have the example $CN_dV_d$



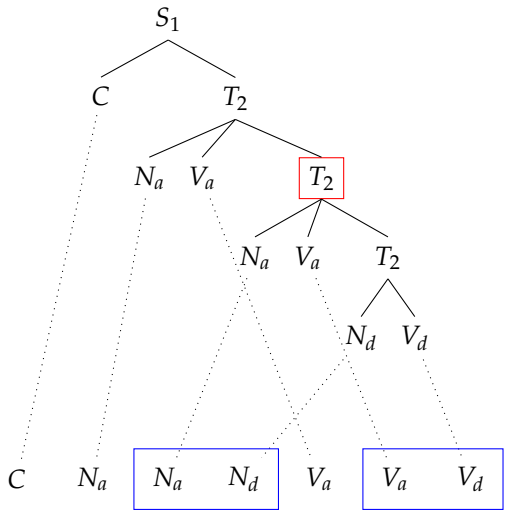
Now lets consider the longer example with three pairs: $CN_aN_aN_dV_aV_aV_d$. We have a derivation tree that looks like the following diagram.



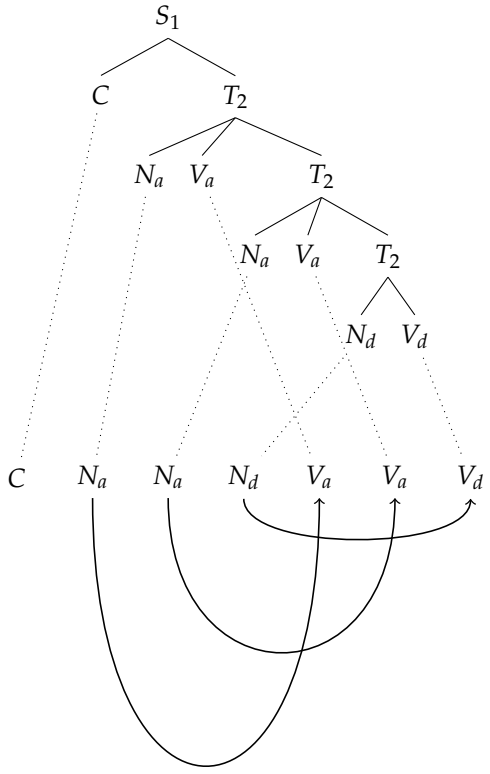

This is a bit cluttered so in the next diagram of the same sentence we have drawn dotted lines from the terminal symbols in the derivation tree to the symbols in their positions in the original string so that the structure of the derivation can be seen more easily.




We also do the same thing but where we mark one of the nonterminals, here boxed in red, and also mark boxed in blue the discontinuous constituent that is derived from that nonterminal.




Finally, by looking at the diagram we can see that we have the right dependencies between the Ns and the Vs: if we look at which ones are derived in the same step, or directly from the same nonterminal, we can see that these correspond to the cross serial dependencies. Reusing the same diagram and adding lines between Ns and Vs that are derived in the same step we get the following situation.



### *Relative clauses*

Let us now try to use MCFGs to do a direct analysis of the relative clause example we looked at before. We repeat the figure here.

The point of the movement analysis is that the word "book" occurs both in the gap, and then in the position as head of the NP, where it is pronounced. We need it to be in both places so it can be analysed in its two semantic roles – a fact that is captured even more directly by the copy theory of movement.

The key insight, which is captured by the Minimalist grammar analysis, is that we don't need to move anything, as long as we derive the word in two places. So this analysis has two parts. The first is that in the position of the gap, we derive the word or phrase that is going to move, but instead of actually concatenating it together, it is held in a waiting position where it is kept until the derivation arrives at the particular point where it is actually pronounced. The second part is where the moving component actually lands, and then it is integrated into the final structure. So in a MG analysis the nodes of the derivation tree have two parts, the part that is actually being built, and a list of zero or more constituents that are in the process of moving.

In the diagram here, we have marked the first part in a blue box, and the second part in a red box. In the blue box in the tree, we see that a transitive verb is combined with the noun "book", however instead of being directly combined into a symbol of dimension 1, we keep it to one side, as it is going to move to another location where it will be pronounced. Higher up in the tree, in the local tree in the red box, we see where the moving constituent lands. Here we have a rule that takes the moving constituent, the $N$ and combines it into the structure. Thus the unary tree in the red box, has a dimension one symbol at the parent, and a dimension 2 symbol as the single child. In between the two boxes we have a succession of rules that resemble context free rules.

We can put this as a value tree in the following example:

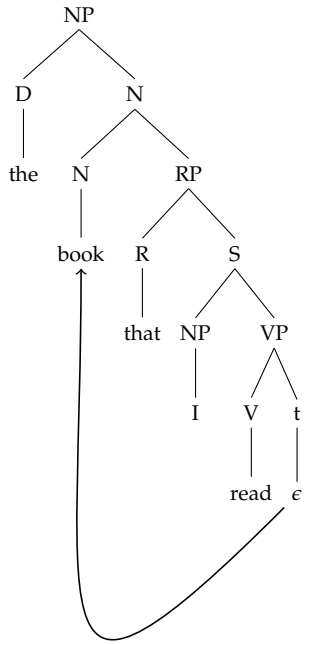
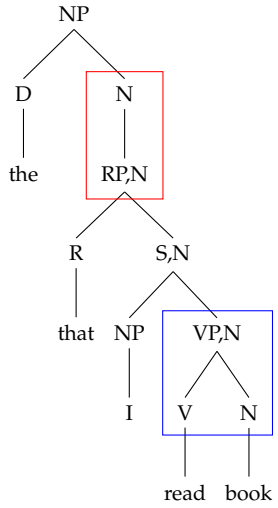
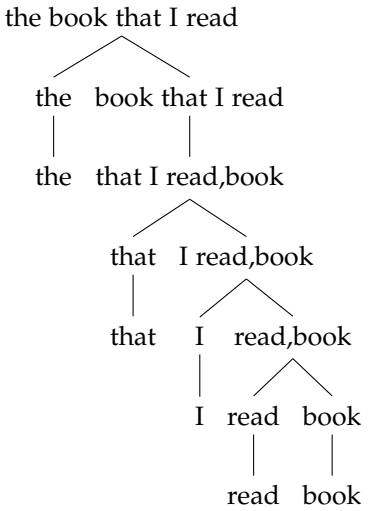



Figure 3: Movement analysis






In fact we will make a minor notational difference and switch the orders of the moving and head labels, so that they reflect the final order in the surface string.

This is because we want to use only non-permuting rules. This means that when we look at Figure4 the diagram on the right hand side has concatenations that do not permute.

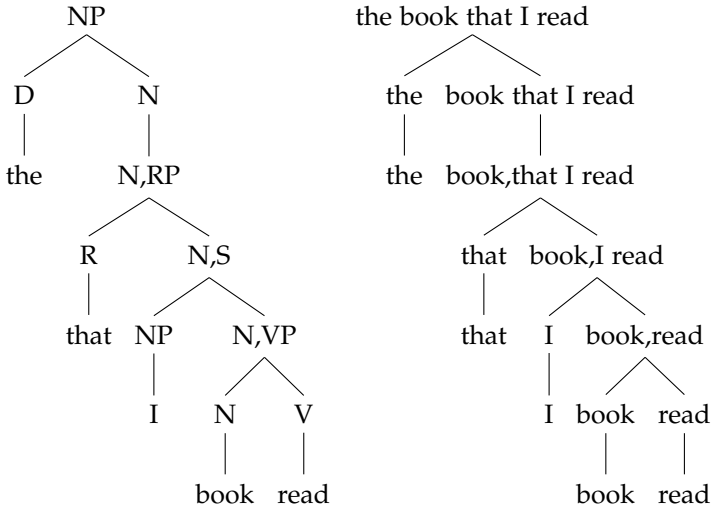



Figure 4: Figure showing the swapped order, with the derivation tree on the left and the value tree on the right.

We can now write down the sorts of rules we need here: for each local tree, assuming that the not very plausible labels on the trees are ok, we write down a rule. The nonterminal symbols here in the MCFG will just be one for each label that we get in the tree. Note that these correspond in some sense to tuples of more primitive labels. Thus the node in the tree labeled $N, S$ will turn into an atomic symbol, $NS$, in the MCFG we are writing, but has some internal structure that we flatten out – it is an $S$ with an $N$ that is moving out of it.

$$NP_1(uv) \leftarrow D_1(u), N_1(v) \tag{22}$$
$$N_1(uv) \leftarrow NRP_2(u, v) \tag{23}$$
$$NRP_2(u, wv) \leftarrow R_1(w), NS_2(u, v) \tag{24}$$
$$NS_2(u, wv) \leftarrow NP_1(w), NVP_2(u, v) \tag{25}$$
$$NVP_2(u, v) \leftarrow N_1(u), V_1(v) \tag{26}$$

Let's look at these rules in a little more detail. The first rule is kind of trivial – this is just a regular CFG rule – all three of the symbols have dimension 1. The second one is more interesting: here we have a symbol with dimension 2 – $NRP_2$. This keeps track of two chunks – the first is the moving chunk which is the $N$, and the second is the chunk that it is moving out of, which is the relative clause. The rule just says – this is the point at which the piece can stop moving, so the concatenation looks trivial. The third (lhs $NRP_2$) is also interesting: this is an example of a much larger class of rules. This rule just transfers the moving constituent up the tree. In a longer example like

(27) This is the book that I told you Bob said I should read.

we will have a long sequence of rules of this type, that pass the moving constituent up. Generally we will have lots of rules like $MX_2(u, wv) \leftarrow Y_1(w), MZ_2(u, v)$. These rules mean that we have a constituent with something moving out of it – this is the $MZ_2$ nonterminal, where the first component $u$ is moving and the second component $v$ means the bit that it is moving out of. We also have a normal constituent without any movement – $Y_1$. The result is a constituent $MX_2$ where the $u$ is still moving. So the $w$ is concatenated onto the $v$ and the $u$ remains as a separate piece. These correspond to rules without a moving constituent like $X_1(wv) \leftarrow Y_1(w), Z_1(v)$.

The symbols $MX_2, X_1$ etc are unrelated in the MCFG formalism but are related at a linguistic level.

Finally we have the rule labeled 26. This rule has a symbol of dimension 2 on the left hand side, and two symbols of dimension 1 on the right hand side. This rule is called when the consituent starts to move. We introduce the noun that is the object of the verb and the verb at the same time, but instead of concatenating them as one might normally, we displace the noun and keep it as a separate component. Crucially the semantic relation between the noun and the verb if local – they are introduced at the same step in the derivation tree, so that when we want to do semantic interpretation they will be close to each other so that the appropriate meaning can be constructed compositionally. In many languages there might also be an agreement relation between the verb and the object, which again could be enforced through purely local rules.

You could also spit this rule into two separate rules using something like a trace or phonetically null element. If we define a null

element using the symbol $T_1$ for trace, we can define a rule $T_1(\lambda)$ that introduces the empty string $\lambda$. We then define a nonterminal of dimension 2, $NT_2$ which generates the moving word and trace. We can then rewrite the rule using the two rules.

$$NVP_2(u_1, vu_2) \leftarrow NT_2(u_1, u_2), V_1(v)$$
$$NT_2(u, v) \leftarrow N_1(u), T_1(v)$$

Such an approach would also be appropriate for languages which have resumptive pronouns which are not phonetically null. In that case we would have a pronounced element instead of $T(\lambda)$.

Now when we consider the MCFG that this is a fragment of, it is clear that it is inadequate in quite a fundamental respect – it fails to capture generalisations between the nonterminal symbols. This is the same limitation that plain CFGs have but it is even more acute now. One of the problems with CFGs that was fixed by GPSG [9] is the fact that there is no relationship between different symbols – either they are identical or they are completely unrelated. In real languages we have, for example, singular NPs and plural NPs. These need to be represented by different nonterminals in a CFG, but there is no way in a plain CFG to represent their relationship. The grammar must tediously spellout every rule twice. Similarly with an MCFG, we have to spell out every combination of rules with moving and nonmoving constituents. The end result will be enormous – finite, of course, but too large to be plausibly learned. Indeed the results that show the equivalence of MCFGs and MGs [10] tend to reveal an exponential blowup in the size of the MCFG that results from the conversion from a Minimalist Grammar. Thus it is important to augment MCFGs with an appropriate feature calculus that can compactly represent the overly large MCFGs, and reduce the redundancies to an acceptable level.

[9] G. Gazdar, E. Klein, G. Pullum, and I. Sag. *Generalised Phrase Structure Grammar*. Basil Blackwell, 1985

[10] J. Michaelis. Transforming linear context-free rewriting systems into minimalist grammars. In P. de Groote, G. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics*, pages 228–244. Springer, 2001

## *Discussion*

The relation to tree adjoining grammars (TAG) and other similar formalisms, such as Linear Context Free Rewriting Systems (LCFRS), is very close. LCFRSs are really just another name for MCFGs, and as Joshi et al. [11] showed, the derivation trees of a wide range of mildly context sensitive formalisms can be modeled by LCFRSs. In a TAG we have an adjunction operation that corresponds to a restricted subset of rules: the derivation trees will use only use well-nested rules. Thus in a TAG we cannot have a rule that looks like this:

$$N_2(ux, vy) \leftarrow P_2(u, v), Q_2(x, y) \tag{28}$$

[11] A.K. Joshi, K. Vijay-Shanker, and D.J. Weir. The convergence of mildly context-sensitive grammar formalisms. In Peter Sells et al., editor, *Foundational Issues in Natural Language Processing*, pages 31–81. MIT Press, Cambridge, MA, 1991

This rule violates the well-nested condition because the $u, v$ pair and the $x, y$ pair will overlap rather than be nested one within the other. Look at the left hand side of this rule and see how the variables from $P_2$ and from $Q_2$ overlap each other. These two rules on the other hand are well nested:

$$N_2(ux, yv) \leftarrow P_2(u, v), Q_2(x, y) \tag{29}$$

$$N_2(xu, vy) \leftarrow P_2(u, v), Q_2(x, y) \tag{30}$$

It is an open question whether non-well-nested rules are necessary for natural language description.

MCFGs in their full generality allow symbols of more than dimension 2. This gives rise to a complicated hierarchy of languages that we will just sketch here. There are two factors that affect this hierarchy – one is the maximum dimension of the symbols in the grammar, and the other is the maximum number of symbols that we have on the right hand side of the rules. In a CFG, we know, since all CFGs can be converted into Chomsky Normal Form, that as long as we are allowed to have at least 2 symbols on the right hand sides of rules, we do not have a restriction. This is not the case for MCFGs. We will call the *rank* of a grammar the maximum number of nonterminals that we have on the right hand side of a rule. There is a complicated 2-dimensional hierarchy of MCFGs based on their rank and dimension. It seems that for natural languages we only need some simple ones at the bottom of the hierarchy, with perhaps dimension and rank only of 2. If we write $r$ for the rank and $d$ for the dimension, then MCFGs can be parsed in time $n^{(r+1)d}$. Note that if we consider CFGs in Chomsky normal form, we have $d = 1$ and $r = 2$, so this gives us a parsing algorithm that is $n^3$ as we would expect. For the case of $r = 2$ and $d = 2$ which may be adequate for natural languages, we need $n^6$.

MCFGs also have a lot of the nice properties that CFGs have — they form what is called an *abstract family of languages*. This means for example that if we have two languages generated by a MCFG then their union can also be defined by an MCFG — they are closed under the union operation. They are also closed under many other operations – intersection with regular sets, Kleene star and so on. This means they are a natural computational class in a certain sense.

They are also not very *linguistic*: you can equally well use these for modelling DNA sequences or anything else. Some of the other formalisms equivalent to MCFGs or subclasses, have quite a lot of interesting linguistic detail baked into the formalism in some way. MCFGs are quite neutral in this respect, just like CFGs. This may or

may not be a good thing, depending on your point of view.

A brief comment about the notation: the form we have for the rules here was introduced by Kanazawa [12] and is in our opinion a great improvement on the original notation of Seki et al.[13]. The notation is perhaps closest to the notation for some other closely related formalisms, such as Range Concatenation Grammars[14] and Literal Movement Grammars [15]: these notations are based on what are called Horn clauses, which are often used in logic programming. We have used $\leftarrow$ instead of the traditional connective $:=$ to emphasize the relation with the derivation operation denoted by $\rightarrow$. The technical literature tends to use the original notation.

If we are interested in cognitive modeling, we are always working at a significant level of abstraction – as Stabler argues [16] we should focus on the abstract thing that is being computed. The equivalence of MCFGs and MGs, as well as the equivalence of subclasses of MCFGs to TAGs and the other well-known equivalences, when taken together, suggest that MCFGs define the right combinatorial operations to model natural language syntax. From a linguistic modeling point of view, MCFGs have two compelling advantages – first they can be efficiently parsed, and secondly various subclasses can be efficiently learned [17].

One of the deep sociological divides in formal/theoretical syntax is between formalisms that use movement and formalisms that don't. One fascinating implication of the research into MCFGs is that it indicates that this divide does not depend on any technical difference. In a certain sense, there is a precise formal equivalence between movement based models and monostratal ones.

[12] Makoto Kanazawa. The pumping lemma for well-nested multiple context-free languages. In *Developments in Language Theory*, pages 312–325. Springer, 2009

[13] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):229, 1991

[14] P. Boullier. Chinese Numbers, MIX, Scrambling, and Range Concatenation Grammars. *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics (EACL 99)*, pages 8–12, 1999

[15] A.V. Groenink. Mild context-sensitivity and tuple-based generalizations of context-grammar. *Linguistics and Philosophy*, 20(6):607–636, 1997

[16] E.P. Stabler. Computational perspectives on minimalism. In Cedric Boeckx, editor, *Oxford Handbook of Linguistic Minimalism*, pages 617–641. 2011

[17] Ryo Yoshinaka. Polynomial-time identification of multiple context-free languages from positive data and membership queries. In *Proceedings of the International Colloquium on Grammatical Inference*, pages 230–244, 2010; and Ryo Yoshinaka. Efficient learning of multiple context-free languages with multidimensional substitutability from positive data. *Theoretical Computer Science*, 412(19):1821 – 1831, 2011

## *Acknowledgments*

Nothing in this note is original; I am merely re-explaining and repackaging the ideas of Makoto Kanazawa, Ed Stabler and many others. My thanks to Ryo Yoshinaka for checking an earlier draft of this note. Thanks also to Chris Brew, Misha Becker for suggesting corrections. There is only a very partial list of citations here: apologies to anyone who feels slighted.

Comments and corrections are very welcome (genuinely — I am not just being polite) and should be sent to `alexsclark@gmail.com`.

## *References*

P. Boullier. Chinese Numbers, MIX, Scrambling, and Range Concatenation Grammars. *Proceedings of the 9th Conference of the European*

*Chapter of the Association for Computational Linguistics (EACL 99)*, pages 8–12, 1999.

N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956.

G. Gazdar, E. Klein, G. Pullum, and I. Sag. *Generalised Phrase Structure Grammar*. Basil Blackwell, 1985.

A.V. Groenink. Mild context-sensitivity and tuple-based generalizations of context-grammar. *Linguistics and Philosophy*, 20(6):607–636, 1997.

A.K. Joshi, K. Vijay-Shanker, and D.J. Weir. The convergence of mildly context-sensitive grammar formalisms. In Peter Sells et al., editor, *Foundational Issues in Natural Language Processing*, pages 31–81. MIT Press, Cambridge, MA, 1991.

L. Kallmeyer. *Parsing beyond context-free grammars*. Springer Verlag, 2010.

Makoto Kanazawa. The pumping lemma for well-nested multiple context-free languages. In *Developments in Language Theory*, pages 312–325. Springer, 2009.

J. Michaelis. Transforming linear context-free rewriting systems into minimalist grammars. In P. de Groote, G. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics*, pages 228–244. Springer, 2001.

P .S. Peters and R. W. Ritchie. On the generative power of transformational grammars. *Information Sciences*, 6:49–83, 1973.

H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):229, 1991.

S. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343, 1985.

E. Stabler. Derivational minimalism. In C. Retoré, editor, *Logical aspects of computational linguistics (LACL 1996)*, pages 68–95. Springer, 1997.

E.P. Stabler. Computational perspectives on minimalism. In Cedric Boeckx, editor, *Oxford Handbook of Linguistic Minimalism*, pages 617–641. 2011.

K. Vijay-Shanker and David J. Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27(6): 511–546, 1994.

Ryo Yoshinaka. Polynomial-time identification of multiple context-free languages from positive data and membership queries. In *Proceedings of the International Colloquium on Grammatical Inference*, pages 230–244, 2010.

Ryo Yoshinaka. Efficient learning of multiple context-free languages with multidimensional substitutability from positive data. *Theoretical Computer Science*, 412(19):1821 – 1831, 2011.