A Language Theoretic Approach to Syntactic Structure

Alexander Clark

Department of Computer Science
Royal Holloway, University of London
Egham, TW20 0EX
United Kingdom
alexc@cs.rhul.ac.uk

Abstract. We consider the idea of defining syntactic structure relative to a language, rather than to a grammar for a language. This allows us to define a notion of hierarchical structure that is independent of the particular grammar, and that depends rather on the properties of various algebraic structures canonically associated with a language. Our goal is not necessarily to recover the traditional ideas of syntactic structure invented by linguists, but rather to come up with an objective notion of syntactic structure that can be used for semantic interpretation. The role of syntactic structure is to bring together words and constituents that are apart on the surface, so they can be combined appropriately. The approach is based on identifying concatenation operations which are non-trivial and using these to constrain the allowable local trees in a structural description.

1 Introduction

When modeling natural languages as formal languages, typically we have a class of representations $\mathcal{R}$ and a function, L, that maps elements of this class to languages. A classic example would be where $\mathcal{R}$ is the class of context free grammars (CFGs), and each CFG, G, defines a context free language (CFL) $L(G)$. Given an element G of $\mathcal{R}$, and a string w, G will typically assign some structural descriptions (SDs) to w. The number may vary – normally if w is not in L we will assign zero SDs, whereas if w is syntactically ambiguous we may want to have more than one SD assigned to w. If G is a context free grammar, then the SDs might be the set of parse trees for w, but other types of grammar define other types of SDs: dependency structures, or labelings of arbitrary trees as in structurally complete variants of categorial grammar. An unambiguous sentence might have only one parse. Alternatively, we might have a formalism where the derivation trees are not directly equivalent to the SDs; such as TAGs or Tree substitution grammars. We can call this approach *the standard model*: a representation defines a set of SDs; each SD defines a unique string. Such a view has been the dominant paradigm in generative linguistics since Syntactic Structures [1]. This

M. Kanazawa et al. (Eds.): MOL 12, LNAI 6878, pp. 39–56, 2011.

leads to the classical distinction between the strong generative capacity of a formalism as the set of SDs that it can generate and the weak generative capacity as the set of strings that it can generate [2].

There is however a problem lurking in this standard model. Since there may in general be several distinct representations that map to the same language, when we consider acquisition, the best we can hope for is that we will learn some grammar which is extensionally (weakly) equivalent to the target. However, this will define a set of SDs that need not be in any way equivalent to the original, target set of SDs; yet we need these SDs in order to support semantic interpretation. This is a version of the famous Argument from the Poverty of the Stimulus [3].

For CFGs the problem is particularly acute – there are infinitely many different CFGs for any non-trivial language, and these can assign essentially arbitrary SDs to any finite subset of strings. The most we can say is that, using a pumping lemma, for sufficiently long strings, there must be some non-terminals that correspond to certain parts of the language.

There are a number of approaches to resolving this problem: one is to take a grammatical formalism which has canonical forms. For example, one could have it that the mapping L is essentially injective, (perhaps up to a change of variables), where all grammars for a given language have the same derivation trees. This is possible for some formalisms (notably for finite automata) but seems hard to do for a sufficiently rich class of languages, or for languages which have the sort of recursive hierarchical structures that we see in natural languages. Another is simply to stipulate that the class is sufficiently restricted in an *ad hoc* way.

Here we take a different approach – rather than having the set of SDs depend on the grammar G for a language L_*, we will have them depend on the language L_* directly. Rather than focusing on the grammar as defining the SDs, we will focus on the SD of a string as being directly generated by the language. A grammar that defines the language will then be able to derive the SDs of the string, not necessarily directly out of the derivation tree, but through some other process. Crucially, two distinct grammars that define the same language will then assign the same (or isomorphic) sets of SDs to strings.

This approach is close in some respects to earlier traditions of linguistics. First it has obvious links with the pre-Chomskyan tradition of American structuralism, for example in the work of Rulon Wells [4]. However these early approaches had some obvious flaws. As Searle [5] puts it:

> How then can we account for these cases where one sentence containing unambiguous words (and morphemes) has several different meanings? Structuralist linguists had little or nothing to say about these cases; they simply ignored them.

The second, less well known tradition, is the Set-theoretical or Kulagina school [6,7]discussed at length in [8]. This large group of researchers, operating in the Soviet Union and Eastern Europe, studied an approach quite similar to this under the name of *configurational analysis*. The particular research program

was not very productive[1], primarily because the technical approaches tried were clearly incorrect, and was largely abandoned by the late 1970s as a result of the critiques of Padučeva, among others.

There are three main problems that a theory of SDs must deal with: the first is the most basic one of dealing with the hierarchical structure of language: this means, informally, that the SD must bring together elements that may be arbitrarily far apart on the surface. The second problem is to deal with ambiguity – both lexical and structural/syntactic. Finally, any theory must deal with the problems of displaced constituents — movement in transformational terminology.

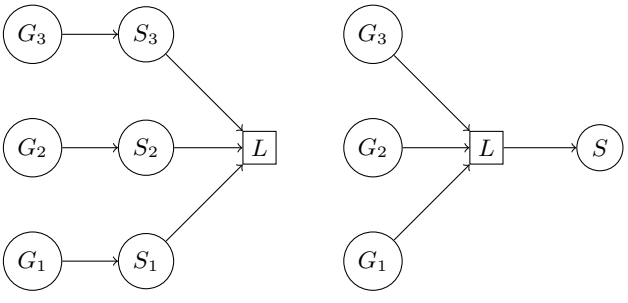


Fig. 1. Diagram comparing grammatical models of SDs with language theoretic ones. L is the language; G_i are grammars and S_i are sets of structural descriptions. On the left we have the standard model: multiple extensionally equivalent grammars, define different sets of structural descriptions. On the right we have multiple grammars, that define the same language which defines a unique set of structural descriptions.

In this paper we will present a simple mathematical model for this as well as various extensions, starting with the most simplest form of distributional model: the congruence class model. After defining the basic ideas of distributional learning in Section 2, we will define the initial model based on congruence classes in Section 3. We then briefly consider two extensions: the first (Section 4) using a generalisation of the congruence relation to tuples of strings, which gives a model similar to Multiple Context Free Grammars (MCFGs), which allows us to handle cross-serial dependencies and displaced constituents, and the second (Section 5) using lattice theory, which allows a better treatment of ambiguity. We then consider some methodological issues in the conclusion.

2 Distributional Learning

We will consider various recent approaches to learning based on ideas of distributional learning. All of these models rely on one basic idea: to look at various

[1] [9] describes it dismissively as "successless".

types of relationships between strings, or tuples of strings, and their contexts. By modeling which combinations of strings and contexts lie in the language, we can construct a family of powerful learning algorithms for various classes of context free and context sensitive languages.

At its most basic level, we consider the relation between a context, defined as a pair of strings (l, r), and a substring u, given by $(l, r) \sim_L u$ iff $lur \in L$. We also consider the natural extension of this to tuples of strings and their generalised contexts $(l, m, r) \sim_L (u, v)$ iff $lumvr \in L$.

The first model [10,11,12], which we call *congruential*, relies on identifying the equivalence classes under complete substitutability. This is close to the classic conception of distributional learning considered by the American structuralists [13]. We also consider the extension of this to equivalence classes of pairs of strings [14]. Finally we use formal concept analysis [15] to construct a hierarchy or lattice of distributionally defined classes [16,17,18].

All of these approaches rely on identifying parts of an algebraic structure underlying the language. The first relies on identifying the syntactic monoid, and the second on identifying the syntactic concept lattice. Both of these structures are associative — in the simple mathematical sense that their concatenation operation, $\circ$, satisfies $X \circ (Y \circ Z) = (X \circ Y) \circ Z$. This is in direct contrast to the fundamentally non-associative operation of tree conjunction: $X(YZ)$ is not the same tree as $(XY)Z$. Thus, at a first glance, the representational assumptions of these theories, based on abstractions of string substitution, seem profoundly incompatible with the hierarchical tree based representations hypothesized in linguistics. The main contribution of this paper is to show that this can be resolved.

2.1 Notation

We now define our notation; we have a finite alphabet Σ; let Σ^* be the set of all strings (the free monoid) over Σ, with λ the empty string. A (formal) language is a subset of Σ^*.

Definition 1. *We will consider various simple formal languages that have various types of dependency.*

- ALL Σ^*. *This language intuitively has no structure.*
- ANBN $\{a^n b^n | n \geq 0\}$. *This language is a linear, non-regular language.*
- DYCK *The Dyck language. These languages are crucially important as, through the Chomsky-Schutzenberger theorem they in a sense encapsulate all possible types of hierarchical structure. Note also that the hierarchical structures are not binary branching: consider for example* ()()(). *Here we replace* (*by* a, *and*) *by* b *for formatting reasons; so this example would be ababab. We will consider the* DYCK2 *language to be the Dyck language of order 2, with* a, b *as open brackets and* c, d *as the corresponding close brackets.*
- EQ $\{w | |w|_a = |w|_b\}$. *The language with equal numbers of* as *and* bs *– this is a non-regular context free language that does not have any intrinsic hierarchical structure.*

- COPY $\{cwcw|w \in (a|b)^*\}$
- AMBDYCK *Ambiguous version of Dyck language. This language has three symbols* $\{a, b, c\}$ *where* a *is open bracket,* b *is closed bracket and* c *can be either. More formally this is generated by the grammar* $S \to \lambda, S \to SS, S \to ASB$ *together with the rules* $A \to a, A \to c, B \to b, B \to c$.
- DISPLACE *A version of the Dyck language of order 2, where one letter may be displaced to the front; we define this more precisely later.*

We can concatenate two languages A and B to get $AB = \{uv|u \in A, b \in B\}$. We will write ordered pairs as (l, r), and $A \times B$ for the Cartesian product of two sets defined as $\{(u, v)|u \in A, v \in B\}$.

A context or *environment*, as it is called in structuralist linguistics, is just an ordered pair of strings that we write (l, r) where l and r refer to left and right; l and r can be of any length. We can combine a context (l, r) with a string u with a wrapping operation that we write $\odot$: so $(l, r) \odot u$ is defined to be lur.

For a given string w we can define the *distribution* of that string to be the set of all contexts that it can appear in: $C_L(w) = \{(l, r)|lwr \in L\}$, equivalently $\{f|f \odot w \in L\}$. Clearly $(\lambda, \lambda) \in C_L(w)$ iff $w \in L$.

Two strings, u and v, are *congruent* with respect to a language L, written $u \equiv_L v$ iff $C_L(u) = C_L(v)$. This is an equivalence relation and we write $[u]_L = \{v|u \equiv_L v\}$ for the equivalence class of u.

Example 1. Suppose the language is ANBN. There are an infinite number of congruence classes. $[a] = \{a\}$ and $[b] = \{b\}$, and similarly we have $[a^k] = \{a^k\}$, and $[b^k] = \{b^k\}$, which are all singleton sets. We have $[\lambda] = \{\lambda\}$ and $[ab] = \{a^i b^i | i > 0\}$. Finally we have $[a^k b] = \{a^{k+i} b^{i+1} | i \geq 0\}$ and $[ab^k] = \{a^{i+1} b^{k+i} | i \geq 0\}$ for every $k > 0$. Note that $L = [\lambda] \cup [ab]$.

It is clear that this relation is a congruence of the monoid: for all strings u, v, w if $u \equiv_L v$ then $uw \equiv_L vw$ and $wu \equiv_L wv$. As a result:

Lemma 1. *For any language* L, *if* $u \equiv_L u'$ *and* $v \equiv_L v'$ *then* $uv \equiv_L u'v'$; *equivalently* $[u]_L[v]_L \subseteq [uv]_L$.

This means that the syntactic monoid $\Sigma^*/\equiv_L$ is well defined, and we can take $[u] \circ [v]$ to be defined as $[uv]$. Note here a crucial difference between set concatenation $[u][v]$ and concatenation in the syntactic monoid $[u] \circ [v]$; the latter may be larger.

Example 2. Suppose the language is ANBN. $[a] = \{a\}$ and $[b] = \{b\}$ so $[a][b] = \{ab\}$, a singleton set. But $[a] \circ [b] = [ab] = \{a^i b^i | i > 0\}$, which is an infinite set, which clearly properly includes $\{ab\}$.

Some distributional learning algorithms for context free grammars [10,14] define non-terminals where each non terminal corresponds to a congruence class. These "congruential" grammars then rely on Lemma 1 to define rules of the form $[uv] \to [u][v]$. The lemma guarantees that $[uv] \supseteq [u][v]$, and that therefore such rules are in a technical sense "correct".

A natural question is when does $[u] \circ [v] = [u][v]$? If this is the case, then in a sense the rule $[uv] \to [u][v]$ is vacuous: it reduces to simple set concatenation. If, on the other hand, $[uv]$ properly includes $[u][v]$ then this is significant in some sense. This distinction is the crux of our approach. We will define this formally below – here we just give an illustrative example.

Example 3. Consider DYCK. The congruence classes are all of the form $[b^i a^j]$ where i, j are non negative integers. $[\lambda] = L$. Concatenations like $[b^i a^j][b^k a^l]$ are non-vacuous when j and k are both nonzero: in particular $[a][b]$ is non vacuous since $[\lambda] = [ab]$ and so λ is in $[ab]$ but not $[a][b]$. If either j or k are zero, then it is vacuous. In particular, $[a^i][a^j]$ are vacuous, as are $[b^i][b^j]$ and $[b^i][a^j]$. For example, $[a] = LaL$ so $[a][a] = LaLLaL = LaLaL = [aa]$.

3 Congruential Representation

We now define the basic structural descriptions that we use. We will consider ordinal trees: singly rooted trees where the child nodes of each node are ordered. Each node in the tree will be labeled with a set of strings. The leaf nodes will be labeled with singleton sets of elements of Σ: individual letters. We will sometimes write these as $\{a\}$, but often we will omit the brackets. The yield of a node in the tree is defined to be the concatenation of the letters at the leafs. Each non leaf node will be labeled with the congruence class of the yield of that node. We will call such a tree *congruentially labeled*.

So, given a string *abc* we have a number of trees, some of which are shown in Figure 2. Note that we always have the trivial tree which has only one non-leaf node. We will not consider trees for the empty string in this paper.

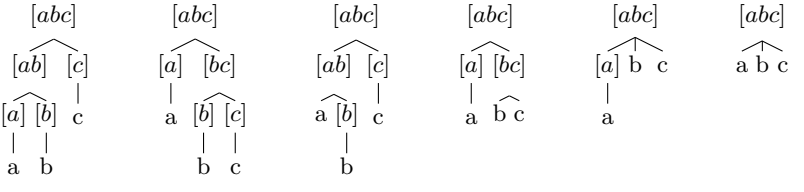


Fig. 2. Some congruentialy labeled trees for *abc*

First we note a basic property of these trees. For every local tree in a tree, the set labeling the parent of the local tree contains the concatenation of the sets labeling the children. This is because of the fact that $\{a\} \subseteq [a]$ and by Lemma 1. We can state this as a lemma without proof.

Lemma 2. *If we have a local tree with parent labeled R and k children labeled $D_1 \dots D_k$ then $R \supseteq D_1 \dots D_k$.*

Consider the following notion of the "cut" of a tree: we take a connected subtree of the graph that contains the root node, and such that if one child of a node is in the subtree then all children of the node are in the subtree. A cut will be the sequence of leaves of such a subtree. If we take any cut of the tree we can form a set of strings by concatenating the labels: every cut will give a subset of the congruence class of the yield. At the root this is the entire congruence class of the yield, at the leaves it is just the singleton set containing the yield, and as the cut moves up (as the subtree gets smaller) the set gets larger monotonically by Lemma 2.

On its own this tree representation is not restrictive. Every string will have every possible tree. Intuitively we want to say that for some languages, some trees are disallowed.

3.1 Definition

Definition 2. *A local tree with parent labeled R and k children labeled $D_1 \dots D_k$ is* **vacuous** *if $R = D_1 \dots D_k$.*

The intuition behind this definition is this: given a local tree, we have the parent labeled with a set of strings, and the children all labeled with sets of strings. If the concatenation of all of the sets of strings is equal to the set of strings in the parent, then the tree structure adds nothing and is vacuous. If on the other hand the parent set is strictly larger then we have a non-trivial combination.

Example 4. Consider again DYCK. The local tree $\widehat{[aa]\ [a]\ [a]}$ is vacuous as $[aa] = \{LaLaL\} = [a][a]$. On the other hand, the local tree $\widehat{L\ [a]\ [b]}$ is not vacuous since $L = [ab] = [\lambda]$ contains λ which is not in $[a][b]$. Unary trees, trees with only one child will always be vacuous unless the child is a leaf and $[a]$ is larger than $\{a\}$. So $[u]$ is clearly vacuous, but $[a]$ is not vacuous, since $aab \in [a]$. A less natural example is this tree $\overset{\frown}{\text{a b a b}}$ which is also not vacuous.

The final tree illustrates that this constraint – that every local tree is non-vacuous – is not enough. We also need to have a constraint that says that the tree is as deep as possible, otherwise we will always have a simple flat structure for every string.

Definition 3. *A local tree with parent R and k children labeled $D_1 \dots D_k$, which is not vacuous, is* minimal *if there is no subsequence of children $D_i D_{i+1} \dots D_j$, $j \geq i$ such that a local tree $[D_i D_{i+1} \dots D_j] \neq D_i \dots D_j$.*

Example 5. Consider again DYCK. L is not minimal since $[ab] \neq \{ab\}$.

L is minimal.

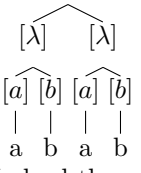
L is not minimal since $[a] \neq a$.

Given these conditions we can now define a condition on trees, that will define the set of SDs for a given string in a language.

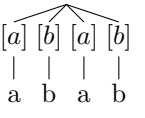
Definition 4. *A congruentially labeled tree is* valid *if and only if every non-root local tree is not vacuous and is minimal. We allow however the root tree to be vacuous, but if it is not vacuous, then it must be minimal.*

We could eliminate the condition on the root tree by assuming that each string in L is bracketed by a distinguished start and end symbol.

Example 6. Consider DYCK. This is a valid tree for *abab*: with a vacuous root node.



On the other hand

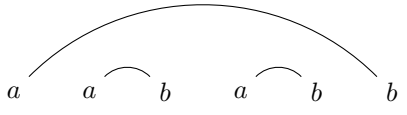


is not valid as it is not minimal. Indeed there is only one valid tree for this string.

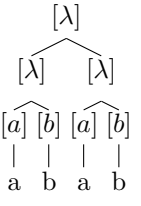
Lemma 3. *For the Dyck language, for every string in L, there is a* unique *valid tree where the non leaf nodes are labeled only with $[\lambda], [a], [b]$. Matched pairs of opening and closing symbols are always in the same local tree.*

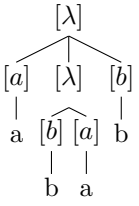
Proof. Looking at the leafs of valid trees, since $[a]$ and $[b]$ are infinite, all letters will be introduced by a pre-terminal node – that is to say, a node with only one child, which can only be non-vacuous if that child is a leaf. Next, note that we can never have a node labeled with $[a^i]$ or $[b^i]$ when $i > 1$ since local trees $[aa] \rightarrow [a][a]$ are vacuous. Note also that we cannot have a node labeled with $[ba]$ since $[ba] \rightarrow [b][a]$ is vacuous. Valid local trees are of the form $L \rightarrow [a][b]$, $L \rightarrow [a]L[b]$, $L \rightarrow [a]L^i[b]$.

Note that the structural descriptions do not correspond to the set of parse trees of any CFG, since the set of valid local trees is infinite for this language. An important consequence is thus locality: matching open and closing brackets are always in the same local tree. There are no other redundant arbitrary local trees. We can thus say, in this trivial example that we have a dependency between two letters, when the leaves occur in the same local tree. We can represent this diagrammatically for the string *aababb*:

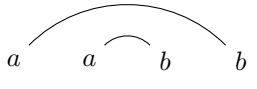


Example 7. Consider EQ. Writing local trees to save space as $[xy] \to [x][y]$, we can see that $[aa] \to [a][a]$ is vacuous, as is $[aab] \to [a][ab]$ but $[ab] \to [a][b]$ is not. So for something like *abab* we have several different structures. Intuitively, we could have a dependency between the first pair, or between the first and the last:

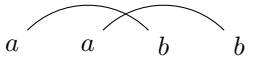




Consider however the string *aabb* in the language EQ. In this example, we slightly extend our notion of dependencies since here we have $[a]$ in the same local tree rather than the letter itself. This only has one SD, which gives the dependencies



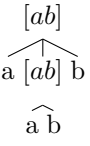
but no SD that gives



So in this model, because of the tree restriction we miss out some plausible dependencies: namely those where the dependencies cross.

Example 8. Consider ANBN. $[a^i] = \{a^i\}$, $[b^i] = \{b^i\}$, $[\lambda] = \{\lambda\}$. $L = [ab] \cup [\lambda]$. Other congruence classes are of the form $[a^i b]$ and $[ab^i]$, which are infinite. Note that $[a][ab] = [aab]$ but that neither is $[a][b]$ equal to $[ab]$, nor is $[a][ab][b]$ equal to $[ab]$.

Every tree has a unique structure which is linear: it has this form (for *aabb*).



Note that pretheoretically there is no reason to assume, in a string like *aabb*, that the correct dependencies are nested rather than crossing. As we shall see, when we move to a richer model, we get a larger set of possible dependencies in this case.

3.2 Problems

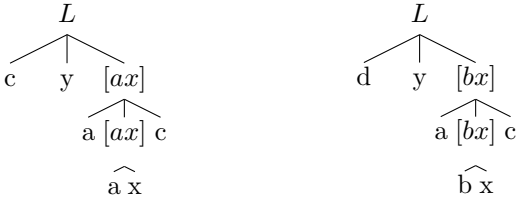
Let us consider some examples where this approach breaks down. If $L = \Sigma^*$, then the only valid trees are ones of depth 1, as all trees will be vacuous. If $L = \Sigma^+$, then every binary branching tree will be valid. This sharp distinction is perhaps undesirable.

Consider the following language, DISPLACE. We start off with DYCK2 – a language over $\{a, b, c, d\}$ where a, b are open brackets and c, d the corresponding close brackets. Given a string w in this language, we consider the following transformation. We take any one occurrence of c or d, and replace it with a new symbol x. The symbol we have removed we place at the front of the string

followed by a symbol y. Think of x as a resumptive pronoun, and y as a relative pronoun. So given a string like *acabdc*, we can pick one of the three occurrences of c or d and get *cyaxabdc*, *cyacabdx* or *dyacabxc*. The shortest strings in this language are *cyax* and *dybx*.

Let us consider this using our simplest model. Note that ax and bx are clearly not congruent, but that ax includes all strings in DYCK2 that are "missing" a c; i.e. $[ax] = \{ax, aacx, aaxc, baxd, \dots\}$.

So we will have trees like



but we will miss the dependency between the x and the c or d at the beginning of the sentence. One approach is to try to have a richer representation, which passes the dependency up through the tree, in a manner similar to GPSG [19].

Consider also AMBDYCK. This is a structurally unambiguous language with some lexical ambiguity. As a result of the lexical ambiguity some strings will have more than one SD. We would like the string *cabc* to have one SD, but the strings *accb* and *cccc* to have two each. This is not possible in this model as all of the concatenations with c are vacuous. This requires the lattice based techniques described in Section 5.

4 Multiple Context Free Grammars

Given the well-known fact that natural languages are not CFLs, it is natural to turn to richer formalisms and representations; just as the theory of distributional learning was lifted from CFGs to Multiple Context Free Grammars (MCFGs) [20] by Yoshinaka [21,22], we can lift the theory of structural descriptions to an MCFG framework easily enough. We assume some familiarity with MCFGs.

We consider now the natural extension of this approach to tuples of strings; for simplicity we will restrict ourselves to the special case of 2-MCFGs, where non-terminals generate either strings or pairs of strings. Given a pair of strings $(u, v) \in \Sigma^* \times \Sigma^*$ we define the natural generalisation of the distribution of this pair in L as in [14]:

$$C_L(u, v) = \{(l, m, r) | l, m, r \in \Sigma^*, lumvr \in L\} \tag{1}$$

We then say that $(u, v) \equiv_L (u', v')$ iff $C_L(u, v) = C_L(u', v')$ and write $[u, v]_L$ for the equivalence class of (u, v) under this relation. We will call this a *bicongruence* class. Note that this is a congruence of the two natural monoids over $\Sigma^* \times \Sigma^*$.

For example consider the COPY language $L = \{cwcw | w \in (a|b)^*\}$. This is clearly not context free. We can see that $[a, a] = \{(a, a)\}$ – no other strings

are congruent to this, but that $(c, c) \equiv_L (ca, ca) \equiv_L (cb, cb)$. Indeed $[c, c]_L = \{(cw, cw) | w \in (a|b)^*\}$.

Let us consider some basic properties of these congruence classes: first, if $u \equiv_L u'$ and $v \equiv_L v'$, we have that $(u, v) \equiv_L (u', v) \equiv_L (u, v') \equiv_L (u', v')$. Therefore $[u, v] \supseteq [u] \times [v]$. More generally $[u, v]$ will be a union of products of congruence classes of strings.

$$[u, v] = \bigcup_{(u', v') \in [u, v]} [u'] \times [v']$$

A natural question to ask is: when is $[u, v] = [u] \times [v]$? In a sense, if this condition holds then we gain nothing by considering the tuple class rather than the original class. If on the other hand, we have it that $[u, v]$ is strictly larger than $[u] \times [v]$ then this is interesting; since $[u, v]$ will always be a union of products of congruence classes, we can think of it as specifying a relation. For example, in the simplest case $[u, v]$ might be $[u] \times [v] \cup [u'] \times [v']$. This means that if we have a string in $[u]$ in one place, then we must have a string in $[v]$ in the other place, and if we have a string in $[u']$, similarly we must have a string in $[v']$: this means that there is a relation between the strings we substitute for u and the strings we must substitute for v to maintain grammaticality.

Definition 5. *Given a language L, we say that a bicongruence class $[u, v]_L$ is trivial iff $[u, v]_L = [u]_L \times [v]_L$.*

It should be noted right now how weak this is – it is only because the language is so trivial that this works. $[a, b]$ just refers to any a followed by any b; we are not restricting them to any structural configuration. It is only because the language is so simple that we know that any a is on the left hand side and any b is on the right hand side that we can have this result. More realistic languages will require a much richer model.

Lemma 4. *All bicongruence classes of* DYCK *are trivial.*

Proof. Suppose that $[b^i a^j, b^k a^l] = [b^w a^x, b^y a^z]$. Note that we will have $(a^i, b^j a^k, b^l)$ in the distribution. and $(a^w, b^x a^y, b^z)$. $a^i b^w a^x b^j a^k b^y a^z b^l \in L$ So we know that $w \leq i$; so by symmetry $w = i$. Similarly we can argue that $j \leq x$, and $x \leq j$ so $x = j$. etc.

Example 9.
- If $L =$ ANBN, note that $(a, b) \equiv_L (aa, bb)$. So $[a, b] = \{(a^i, b^i) | i > 0\}$ which is clearly larger than $[a] \times [b]$.
- Consider EQ: here $[a, b] \equiv_L [ab, ab]$ so again this is non trivial.
- Consider COPY: $[c, c] = \{(cw, cw)\}$ which is non-trivial, but $[a, a] = \{(a, a)\}$ which is trivial.

We can now start to define the appropriate tree structures. First we define a class of functions, from tuples of tuples of strings to tuples of strings. These are the sorts of functions used in the definition of MCFGs; we add some standard restrictions. We write the class of functions of arity $i \to j_1, \ldots j_k$ to be $\mathcal{F}_{i \to j_1, \ldots j_k}$. Each

function of the type that we are concerned with can be written as i sequences of indices: We write an index as an ordered pair of positive integers $[m, l]$: intuitively this means the mth part of the lth component, where m is $1, \ldots, j_l$ and l is in $1 \ldots k$. We require that each index occur exactly once, and furthermore that if $m \leq m'$ then $[m, l]$ must occur before $[m', l]$. Here we only consider pairs of strings so m is at most 2. We may have trees that are non binary so l could be larger than 2.

Simple string concatenation is in $\mathcal{F}_{1\to 1,1}$ and is written as $\langle [1,1][1,2] \rangle$.

We will write tuples of various arities as $\mathbf{t}$, and we will overload $\equiv_L$. We can now extend the functions in $\mathcal{F}$ to sets of tuples, in the natural way: e.g. $f(\mathbf{T}_1, \mathbf{T}_2) = \{f(\mathbf{t_1}, \mathbf{t_2}) | \mathbf{t_1} \in \mathbf{T}_1, \mathbf{t_2} \in \mathbf{T}_2\}$.

Note that because of our construction it is the case that for all functions $f \in \mathcal{F}$, if $f \in \mathcal{F}$, and if $\mathbf{t_i} \equiv_L \mathbf{s_i}$ then $f(\mathbf{t_1}, \ldots, \mathbf{t_k}) \equiv_L f(\mathbf{s_1}, \ldots, \mathbf{s_k})$. That is to say, $[f(\mathbf{t_1}, \ldots, \mathbf{t_k})] \supseteq f([\mathbf{t_1}], \ldots, [\mathbf{t_k}])$, by a simple extension of Lemma 1: see [14] for proof. Alternatively we can see this as being the claim that the relation $\equiv_L$ is a congruence with respect to the functions in $\mathcal{F}$.

Again we can consider cases where the concatenation is vacuous:

Definition 6. *If we have a function $f \in \mathcal{F}$ of various tuples $\mathbf{t_1}, \ldots, \mathbf{t_k}$, we say it is* vacuous *if $[f(\mathbf{t_1}, \ldots, \mathbf{t_k})] = f([\mathbf{t_1}], \ldots, [\mathbf{t_k}])$.*

Example 10. Consider again EQ: if f is $\langle [1,1][1,2], [2,1][2,2] \rangle$, then $f((a,b),(a,b))$ $= (aa, bb)$ is vacuous since $[aa, bb] = \{(u,v) | uv \in L\} = [\lambda, \lambda]$, which is equal to $f([\lambda, \lambda], [\lambda, \lambda])$.

Definition 7. *A non-vacuous function $f \in \mathcal{F}$ of various tuples $\mathbf{t_1}, \ldots, \mathbf{t_k}$, is* **minimal** *if for any two functions $g, h \in \mathcal{F}$ such that $f(\mathbf{t_1}, \ldots, \mathbf{t_k}) = g(\mathbf{t_1}, \ldots, \mathbf{t_J}$, $h(\mathbf{t_{j+1}}, \ldots, \mathbf{t_l}), \mathbf{t_{l+1}}, \ldots, \mathbf{t_k})$, at least one of g and h are vacuous. If we can find a pair of non-vacuous such functions, then f is not minimal.*

Note that if all tuples are of arity 1, then this definition coincides with the definition earlier in Section 3.

Now we define the corresponding notion of a tree. We have a tree, where the leafs are labeled with a singleton set of a letter as before. We label each non-leaf node of the tree with a function, which determines how the string or string tuple corresponding to that node is formed from the strings or tuples corresponding to the child nodes. The arity of the functions must be compatible: we say that the arity of a node is the arity of the output of the function, and in a local tree with children $t_1, \ldots, t_k$ with arities $a_1, \ldots a_k$ and where the tree is labeled with a function of arity a_0, the function must be of arity $a_0 \to a_1, \ldots, a_k$. This condition merely states that the functions must be of appropriate types given that some nodes are labeled with strings and some with tuples of strings. Clearly this also means that some arities are disallowed – if we have a unary local tree where the child has arity one and the parent has arity two, there is no string that can fill the second slot of the parents label, and thus this situation is impossible.

We then define the string function: we recursively compute a string that corresponds to each node, using the functions, and proceeding bottom up from the leaves. The yield of the tree is the string labeling the root. We stipulate that the root node, as with the leaf nodes, must have arity 1. Each node is then labeled with a string or pair of strings, which is formed by the application of the function to the strings or tuples of strings at each child node. We say that it is a tree for a string w, if the string labeling the root is w. Since all of the leafs contain single letters of w, and because of the restrictions of the rules, it is clear that, in a tree for a string w, there will be a bijection between the leafs of the tree and the occurrences of letters in w.

The set of valid trees for a string w will then be the set of trees where each local tree is non vacuous and minimal, with the exception as before for the root node. We will illustrate this definition with some simple examples.

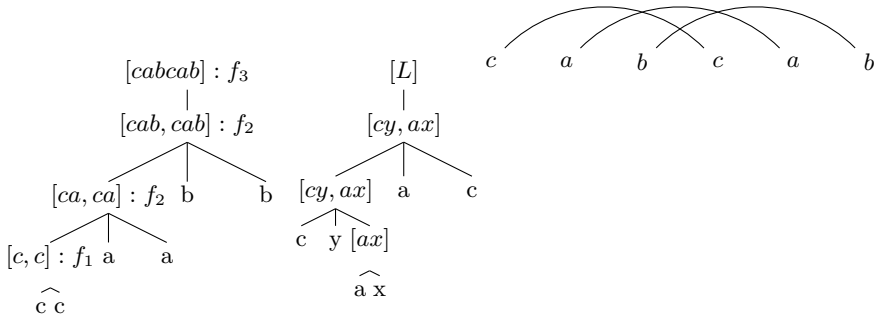


Fig. 3. Trees from Examples 11 and 12

Example 11. Consider COPY: we only have one structure for strings in the copy language. So for *cabcab*, given the functions $f_3 = \langle [1,1][1,2] \rangle$, $f_2 = \langle [1,1][1,2], [2,1][1,3] \rangle$ and $f_1 = \langle [1,1], [1,2] \rangle$, we have the tree on the left of Figure 3. Note that the non root local trees are non vacuous: taking the middle one: $[f_2([c,c], a, a)] = \{(cwa, cwa) | w \in (a|b)^*\}$, whereas $[ca, ca] = \{(cw, cw) | w \in (a|b)^*\}$ which is strictly larger. This gives us the dependencies shown on the right of Figure 3.

Example 12. Consider DISPLACE. We have $[cy, ax] = [dy, bx]$ which is therefore not trivial, and we have non vacuous local trees that combine with this. This gives us some trees for *cyaaxc* like the one in the middle of Figure 3.

Again, please note that the presence of the "resumptive pronoun" x and the relative pronoun y and the fact that we only have one clause is vital to this very simple model working. Nevertheless, this richer model, based on the theory of congruential MCFGs provides a possible solution both for representing cross serial dependencies, and for representing displaced constituents.

5 Lattice Approaches

The approaches based on congruence classes are too restrictive since exact equality of distribution is too strict a requirement. For the simple artificial examples we consider here it is mostly adequate, but for natural languages it is too simplistic [23]. In English, for example, it is quite rare to find two words that are exactly substitutable. "cat" does not have quite the same distribution as "dog" – consider the phrase "to dog someone's footsteps" – any two verbs that have slightly different sets of subcategorisation frames will not be congruent, and so on. The problems are only magnified when we move to the phrasal level. The important point is that though "cat" and "dog" may have distributions that differ slightly, they nonetheless overlap to a great extent. We thus need a mechanism that can represent the shared distribution of two or more words: a more abstract representation that can work with classes that are larger than the congruence classes. Indeed congruence classes are the smallest possible sets that can be defined distributionally.

It is therefore natural to move to a representation that uses larger classes. We therefore look to the lattice based approaches described in [24,17]. Rather than considering just the congruence classes, we consider the Galois lattice formed from the context-substring relation. Given a set of strings S we can define a set of contexts S' to be the set of contexts that appear with every element of S.

$$S' = \{(l, r) : \forall w \in S \;\; lwr \in L\} \tag{2}$$

Dually we can define for a set of contexts C the set of strings C' that occur with all of the elements of C

$$C' = \{w : \forall (l, r) \in C \;\; lwr \in L\} \tag{3}$$

A set of strings S is defined to be closed iff $S'' = S$. Given any set of strings S, S'' is always closed. For those familar with these lattice techniques, a set S is closed iff $\langle S, S' \rangle$ is a concept. Here we consider only the set of strings rather than a pair of strings and contexts, for notational continuity with the preceding sections. Each closed set of strings will be a union of congruence classes. If $S = \{w\}$, then S'' will include $[w]$ but also any other strings u where $C_L(u) \supseteq C_L(w)$. Note that L is always a closed set, since $(\lambda, \lambda) \in S'$.

We will therefore consider trees as before, but where each non leaf node is labeled with a closed set of strings that includes the yield of the subtree. The leaf nodes will be labeled as before with singleton sets of letters. That is to say, rather than the set of labels being drawn from congruence classes, they are drawn from the set of closed sets of strings. It is now possible that there will be more than one possible label for each node since there may be many closed sets of strings that contain a given string.

Example 13. Consider ANBN. There are 3 closed sets of strings that contain a: these are $\{a\}$, $\{a^{i+1}b^i | i \geq 0\}$ and Σ^*.

Definition 8. *A tree is* lattice labeled *if every non leaf node in the tree is labeled with a closed set of strings, and every leaf node is labeled with a singleton letter, and they satisfy two conditions: first each non leaf node is labeled with a set that contains the concatenation of the labels of the child nodes; secondly, the root node must be labeled with a subset of* L.

We can now modify the conditions on trees to reflect the different set of labels. First of all we require that each local tree have the set at the parent *properly* containing the concatenation of the sets at the children. Secondly, as before we have a minimality condition, which we formulate identically. This will give us a set of representations that will include the congruentially labeled trees as before, with the minor change that each node with yield w will be labeled with $\{w\}''$ rather than $[u]$. We can get a more interesting set of trees by adding one more condition that the label at each node is as large as it can be.

Definition 9. *A non leaf node in a lattice labeled tree is* **maximal** *if it cannot be replaced with a larger set, without violating the lattice labeling conditions in Definition 8. We then say that a valid tree will be a lattice labeled tree where every non leaf node is maximal.*

Note that as a result, every valid tree will have a root labeled L.

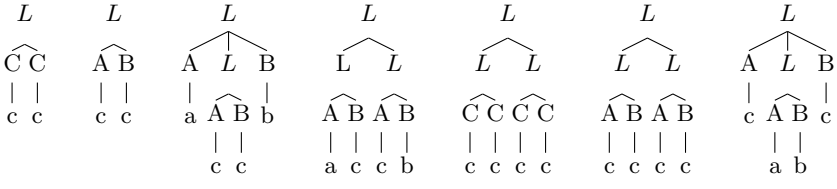


Fig. 4. Some lattice labeled trees. From left to right we have an invalid tree for *cc* and then a valid one. We then have two valid trees for *accb*, and then an invalid tree for *cccc* followed by one of the two valid trees for this string. Finally we have the unique tree for *cabc*.

We will now consider AMBDYCK. The lattice structure of this language is rather complicated. We will just consider a few closed sets, $A = \{a\}''$, $B = \{b\}''$, $C = \{c\}'' = \{c, cab, ccc, \dots\}$. Note that $C \subset A$ and $C \subset B$. We also have L, and $C^2 = \{cc\}''$ where $C^2 \subset L$. Figure 4 illustrates some lattice labeled trees for this language.

6 Discussion

We have presented a basic model and two extensions to deal with displacement, cross-serial dependencies and ambiguity. It is possible to combine the two: to have a lattice of tuples of strings. Note that this idea of structural descriptions

differs in several respects from the traditional derivation trees. The notions of ambiguity that we use are unrelated to the idea of inherently ambiguous CFGs. Indeed the inherent structure of the DYCK language shows the unsuitability of CFG derivation trees for modelling syntactic structure, though this can of course be dealt with by the notational extension of allowing regular expressions on the right hand sides of productions [25].

This paper is largely mathematical: we do not consider any algorithms in this paper but it is natural to consider using congruential learning algorithms to learn a congruential CFG/MCFG for a language and then to parse using that representation. The example of DYCK shows that the parse trees will not directly correspond to the rules, but every parse tree with respect to a congruential CFG will be a congruentially labeled tree: it will not in general be valid. Space does not permit a full exploration of these possibilities. There are two main problems that need to be dealt with: testing to see whether a local tree is vacuous, and coping with the possibly exponential number of parse trees for a given string in a compact way. Appropriate dynamic programming algorithms must be left for future work. We also do not consider any language theoretic issues. It is natural to consider the relation between these trees and the class of congruential context free grammars (CCFGs). For example, take the class of all languages that have the following property: there is a finite set of congruence classes K such that every valid tree for every string in L has labels drawn only from K. What is the relation between this class and the class of CCFGs?

How do the traditional linguistic ideas of constituent structure compare to the ideas we present here? First note that the now standard ideas of constituent structure [26] arose out of exactly the distributional learning algorithms from structuralist linguistics that we are operationalising here [27, p.172, fn 15]. There are standard tests for constituent structure: these include substitutability, the existence of pro-forms, coordination tests and the like. The approach we present here does not explicitly use these properties, except of course the distributional criterion of substitutability. However, if there is a pro-form, then this means that the local tree is likely to be non vacuous, and thus these algorithms implicitly exploit the same information. Therefore we can derive these tests from a deeper principle. Rulon Wells [4] says:

> What is difficult, but far more important than either of the easy tasks, is to define focus-classes rich both in the number of environments characterizing them and at the same time in the diversity of sequence classes that they embrace.

This is almost exactly the point: it is only a local tree where the parent has a diverse collection of "sequence-classes", in the sense that they are not characterised by a single sequence of classes, that will be non vacuous.

A central question is whether distributional learning methods are rich enough to acquire a suitable notion of syntactic structure from raw strings. In other words, is it possible to go from strings to trees, and still richer structures without an external source of information, such as semantic or prosodic structure? The present approach seems to indicate that at least for some classes of

languages, which include many simple examples, this is possible. A second question is whether a purely local concatenation operation is sufficient, or whether it is neccessary to allow non-local operations: in Minimalist Program terms, is external MERGE enough or do we need internal MERGE/MOVE?

We close with an often misquoted passage from Book VIII of Aristotle's Metaphysics:

> In the case of all things which have several parts and in which the totality is not, as it were, a mere heap, but the whole is something beside the parts, there is a cause;

In the context of this paper we can say that where the whole is greater than the concatenation of the parts, we must look for a cause; and that cause is syntactic structure.

Acknowledgments. We are grateful to the reviewers for helpful comments.

References

1. Chomsky, N.: Syntactic Structures. Mouton, Netherlands (1957)
2. Miller, P.: Strong generative capacity: The semantics of linguistic formalism. CSLI Publications, Stanford (1999)
3. Clark, A., Lappin, S.: Linguistic Nativism and the Poverty of the Stimulus. Wiley-Blackwell (2011)
4. Wells, R.S.: Immediate constituents. Language 23(2), 81–117 (1947)
5. Searle, J.R.: Chomsky's revolution in linguistics. The New York Review of Books 18(12) (June 1972)
6. Kulagina, O.S.: One method of defining grammatical concepts on the basis of set theory. Problemy Kiberneticy 1, 203–214 (1958) (in Russian)
7. Sestier, A.: Contribution à une théorie ensembliste des classifications linguistiques. In: Premier Congrès de l'Association Française de Calcul, Grenoble, pp. 293–305 (1960)
8. van Helden, W.: Case and gender: Concept formation between morphology and syntax (II volumes). Studies in Slavic and General Linguistics. Rodopi, Amsterdam-Atlanta (1993)
9. Meyer, P.: Grammatical categories and the methodology of linguistics. Russian Linguistics 18(3), 341–377 (1994)
10. Clark, A., Eyraud, R.: Polynomial identification in the limit of substitutable context-free languages. Journal of Machine Learning Research 8, 1725–1745 (2007)
11. Clark, A.: Distributional learning of some context-free languages with a minimally adequate teacher. In: Sempere, J.M., García, P. (eds.) ICGI 2010. LNCS, vol. 6339, pp. 24–37. Springer, Heidelberg (2010)
12. Clark, A.: PAC-learning unambiguous NTS languages. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 59–71. Springer, Heidelberg (2006)
13. Chomsky, N.: Review of Joshua Greenberg's Essays in Linguistics. Word 15, 202–218 (1959)

14. Yoshinaka, R., Clark, A.: Polynomial time learning of some multiple context-free languages with a minimally adequate teacher. In: Proceedings of the 15th Conference on Formal Grammar, Copenhagen, Denmark (2010)
15. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, Heidelberg (1997)
16. Clark, A., Eyraud, R., Habrard, A.: Using contextual representations to efficiently learn context-free languages. Journal of Machine Learning Research 11, 2707–2744 (2010)
17. Clark, A.: Learning context free grammars with the syntactic concept lattice. In: Sempere, J.M., García, P. (eds.) ICGI 2010. LNCS, vol. 6339, pp. 38–51. Springer, Heidelberg (2010)
18. Clark, A.: Efficient, correct, unsupervised learning of context-sensitive languages. In: Proceedings of CoNLL, Uppsala, Sweden, pp. 28–37 (2010)
19. Gazdar, G., Klein, E., Pullum, G., Sag, I.: Generalised Phrase Structure Grammar. Basil Blackwell, Malden (1985)
20. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science 88(2), 229 (1991)
21. Yoshinaka, R.: Learning mildly context-sensitive languages with multidimensional substitutability from positive data. In: Gavaldà, R., Lugosi, G., Zeugmann, T., Zilles, S. (eds.) ALT 2009. LNCS, vol. 5809, pp. 278–292. Springer, Heidelberg (2009)
22. Yoshinaka, R.: Polynomial-time identification of multiple context-free languages from positive data and membership queries. In: Sempere, J.M., García, P. (eds.) ICGI 2010. LNCS, vol. 6339, pp. 230–244. Springer, Heidelberg (2010)
23. Scholz, B., Pullum, G.: Systematicity and Natural Language Syntax. Croatian Journal of Philosophy 3(21), 375 (2007)
24. Clark, A.: A learnable representation for syntax using residuated lattices. In: de Groote, P., Egg, M., Kallmeyer, L. (eds.) Formal Grammar. LNCS, vol. 5591, pp. 183–198. Springer, Heidelberg (2011)
25. Manaster-Ramer, A.: Dutch as a formal language. Linguistics and Philosophy 10(2), 221–246 (1987)
26. Carnie, A.: Constituent structure. Oxford University Press, USA (2008)
27. Chomsky, N.: Language and mind, 3rd edn. Cambridge University Press, Cambridge (2006)