Is Adjunction Compositional?

Marcus Kracht [*]
Department of Linguistics, UCLA
PO Box 951543
405 Hilgard Avenue
Los Angeles, CA 90095–1543
USA
`kracht@humnet.ucla.edu`

Abstract

This paper shows that there is no compositional TAG for boolean expressions. This indicates that adjunction cannot carry the weight of constructing the semantics compositionally, in contrast to approaches based on discontinuity instead (like Linear Context Free Rewrite Systems). Although the proof is based on the assumption that the semantic functions are partial it seems highly unlikely that allowing partial semantic functions will help.

1 Introduction

In recent years the idea of grammars based on adjunction has received growing attention. Two schools of thought exist which both use the term adjunction: one using tree adjunction (associated with the name Aravind Joshi) and another using string adjunction (associated with the name Solomon Marcus).[1] A grammar

[*]I am indebted to several reviewers for the journal for their criticism and also to Ben George for many conversations about the topic of TAGs and compositionality. Remaining errors are my own responsibility.

[1]Many differences exist, both across and within schools, and it is not possible to do justice to all variants that exist. I have tried to make my results as general as I can.

based purely on adjunction views sentences not as being made from simpler constituents, which are mostly of different nature; rather, it views sentences as being derived from sentences through the insertion of material. Other grammars mix adjunction with tree substitution. Most current versions of TAGs are of such a mixed type. Recently, a series of proposals for computing semantic representations using TAGs has been put forward, of which I mention only [Gardent and Parmentier, 2005], [Gardent and Kallmeyer, 2003] and [Frank and van Genabith, 2001]. [Kallmeyer and Joshi, 2003] even claim to have a compositional semantics.

This paper deals with pure adjunction grammars and the possibility of using them to derive interpreted languages in a compositional fashion. The result will be that for the languages that we are commonly interested in (natural languages, independently motivated formal languages, programming languages) the pure adjunction grammars are unsuited. I hasten to add that this does not mean that the mixed types can make any better use of adjunction. For the results below will indicate that adjunction in general is of quite limited value for interpreted languages, and thus to the extent that the grammars do the job as intended they must rely on substitution instead.

The paper is structured as follows. The first section introduces interpreted languages and sketches how to turn context free grammars into interpreted grammars, that is, grammars for interpreted languages. After that, in Section 3, I shall introduce various kinds of adjunction grammars. Section 4 introduces the notion of interpreted languages, interpreted grammars and compositional grammars. In Section 5 I shall prove some basic results concerning the power of these grammars. The examples are mostly artificial, showing that interpreted adjunction grammars and interpreted context free grammars are not comparable in general. Finally, in Section 6 I shall look at a basic interpreted language, that of boolean expressions. The first negative result concerns unregulated TAGs: they cannot even generate the interpreted language consisting of the variables (paired with their meanings). This is because these grammars completely lack any control over the adjunction sites. Section 7 contains the result that the language of boolean expressions has no compositional TAG based only on adjunction. Section 8 provides a natural language equivalent of the language of boolean expressions. In Section 9 I discuss the consequences for natural language.

2 Context Free Languages

Before we begin the discussion of adjunction, we need to investigate briefly the situation in context free languages (CFLs). Formal details if needed will be supplied in the subsequent sections. CFLs are formulated using rules like the following.

(1) $\mathtt{S} \to \mathtt{AS} \mid \mathtt{AB} \mid \mathtt{c}$

(The vertical bar allows to group together rules with identical left hand side.) If seen as a production device we need to read it from left to right: replace the symbol S in a string either by AS, or by AB, or by c. If no other rule expanding S exists, then this is the only way one can rewrite S. Seen as an analysis rule it is read from right to left: the string c may be seen as a string of category S; if $\vec{x}$ is a string of category A and $\vec{y}$ a string of category S, then $\vec{x}^\frown\vec{y}$ is a string of category S; and so on. Seen in the latter way, the grammar divides the set of terminal strings into classes, one for each nonterminal (plus the class of strings that do not belong to any category). Notice however that the latter description treats the category as implicit, while the first viewpoint treats it as explicit. In the production rules, the nonterminals are parts of the strings, in the analysis rules they are not. This will be important.

Rather than studying string languages, however, we want to study interpreted languages.

Definition 1 (Interpreted Language) *A* ***sign*** *is a pair $\langle \vec{x}, m\rangle$ such that $\vec{x}$ is a string. In this connection, m is the* ***meaning*** *of the sign and $\vec{x}$ its exponent. An* ***interpreted language*** *is a set of signs. Given an interpreted language L, the* ***string language of*** *L is*

(2) $s[L] := \{\vec{x} : \text{ there is } m{:}\langle \vec{x}, m\rangle \in L\}$

A grammar for interpreted languages generates signs in place of strings; it is compositional if the functions act independently on the strings and the meanings. Below we shall study grammars of this form. For interpreted rules we can in fact give two formulations, corresponding to the derivational and the analytic perspective. If we view rules as production devices, we must introduce variables in semantics and view the rules as refinements on their meanings. This however is not the most common interpretation. For semantics, one typically views rules as specifications of how to compose a string (or structure) from simpler ones; given the meanings of the composing elements, the meaning of the composed element is then derived using some function. It is this viewpoint that we take here as well, although nothing hinges on that choice. So, the first of the above rules ($\mathtt{S} \to \mathtt{AS}$) is now rendered as follows. We introduce a binary function ρ, which acts on pairs of signs:

$$(3) \qquad \rho(\langle \vec{x}, m\rangle, \langle \vec{y}, n\rangle) := \begin{cases} \langle \vec{x}^\frown\vec{y}, f(m,n)\rangle & \text{if } \vec{x} \text{ is of category } \mathtt{A} \\ & \text{and } \vec{y} \text{ of category } \mathtt{S} \\ \text{undefined} & \text{else} \end{cases}$$

Here, f is a suitable function that generates the meaning of the complex expression. $\vec{x}^\frown\vec{y}$ will of course be of category $\mathtt{S}$. If several meanings exist we postulate more rules, which act identically on the strings but produce different meanings. Throughout this paper we shall make no assumption on the special character of the semantic functions. Also, notice that the same syntactic operation can be paired with several semantic functions, and conversely a given semantic function with several syntactic operations. Notice also that the functions may be partial, which is one way to account for the different syntactic and semantic categories (see [Kracht, 2006] for a discussion of partiality and categorisation).

As it will turn out in the next section, adjunction grammars do not classify the sets of strings they generate into categories. This is because string adjunction grammars generate only sentential strings. Thus, in order to be able to compare adjunction grammars and CFGs the interpreted languages do not contain any indication of classification. Thus, we are once more led to adopt the analysis viewpoint: rules tell us how plain strings can be composed, that is, strings that do not contain any nonterminals.

Notice, however, that the two viewpoints are not identical. The analysis view cannot dictate, for example, that a given string $\vec{x}$ has meaning m if used as a string of category $\mathtt{A}$, and another meaning m' if used as a string of a different category $\mathtt{B}$. This is a disadvantage when dealing with natural languages where this situation is not uncommon.[2] On the other hand, if we are willing to take a slightly more abstract view on strings (say, if we allow them to contain the category letter or be implicitly typed strings, as proposed in [Kracht, 2003]) these things can be accommodated.

[2]I should stress that despite appearance this problem can easily be dealt with. Categories can be eliminated, see [Kracht, 2007]. But there is, as always, a price to pay.

3 Adjunction Grammars

In contrast to CFGs, adjunction grammars do not generate strings (or structures) of arbitrary kind. Rather, they generate only one type of expression: sentences. The basic mechanism of adjunction grammars is that of *adjunction*. From a semantic viewpoint this contains the message that the only things one needs to specify is sentential meanings and transformations thereof. I am not sure whether this is an advantage, but I shall not comment further on this question. These grammars generate strings (or trees) from other strings (trees) by means of a unary operation of adjunction.[3] Adjunction proceeds by inserting a given pair of strings at two places in the host string. If $\langle \vec{u}, \vec{v} \rangle$ is such a pair, and $\vec{x}$ a string, then we first choose a decomposition

(4) $\vec{x} = \vec{y}_1 \vec{y}_2 \vec{y}_3$

Based on this decomposition the result is

(5) $\vec{y}_1 \vec{u} \vec{y}_2 \vec{v} \vec{y}_3$

This motivates the following definition.

Definition 2 (Sites and Adjunction) *An **adjunction site** is a triple of strings $\sigma = \langle \vec{x}, \vec{y}, \vec{z} \rangle$. We say that $\langle \vec{x}, \vec{z} \rangle$ is the **context** of σ and $\vec{y}$ the **kernel**. A **locale** is a set of adjunction sites. An **adjunction string** is a pair $\alpha = \langle \vec{u}, \vec{v} \rangle$. An **adjunction rule** is a pair $\rho = \langle \alpha, \Lambda \rangle$, where α is an adjunction string and Λ a locale. We write $\vec{p} \to_\rho \vec{q}$ if there is a decomposition $\vec{p} = \vec{x}\vec{y}\vec{z}$ with $\langle \vec{x}, \vec{y}, \vec{z} \rangle \in \Lambda$ and $\vec{q} = \vec{x}\vec{u}\vec{y}\vec{v}\vec{z}$. For a set R of rules we write $\vec{p} \to_R \vec{q}$ if there is a rule $\rho \in R$ such that $\vec{p} \to_\rho \vec{q}$. We write $\to_R^*$ for the reflexive transitive closure of $\to_R$.*

Notice that for any given $\vec{x}$ there may be any number of $\vec{y}$ such that $\vec{x} \to_\rho \vec{y}$. This is because a locale can have any number of sites that decompose $\vec{x}$. (Of course, for $\vec{x}$ of given length n there can be at most $\frac{(n+1)(n+2)}{2}$ many sites, but this number evidently grows as n grows. Thus *any number* means that the number of decompositions is not bounded uniformly for all $\vec{x}$.)

This definition is far more general than used in standard contextual grammars, which we call for want of a distinctive name *factored*. Thus, the standard version of contextual grammar corresponds to what I call a factored contextual grammar. (Cf. among other [Martín-Vide and Păun, 1998] for a definition of contextual grammars.)

[3]The fact that these functions (or rather rules) are unary will have serious consequences.

Definition 3 (Factored Adjunction Rule) *A string adjunction rule $\rho = \langle \alpha, \Lambda \rangle$ is called* ***factored*** *if there exist sets S of strings and C of pairs of strings such that*

(6) $\Lambda = \{\langle \vec{x}, \vec{y}, \vec{z} \rangle : \langle \vec{x}, \vec{z} \rangle \in C, \vec{y} \in S\}$

S is called the ***selector*** *and C the* ***context*** *of ρ.*

The factored grammars require that the kernels and the contexts are specified independently from each other. Unfactored grammars allow to specify pairs of kernel and context on an individual basis, thus allowing rules to be fine tuned to individual strings.

Definition 4 (String Adjunction Grammar) *We call a* ***string adjunction grammar (SAG)*** *a pair $G = \langle B, R \rangle$, such that B is a set of strings, and R a set of adjunction rules. We write*

(7) $L(G) := \{\vec{y} : \text{there is } \vec{x} \in B : \vec{x} \to_R^* \vec{y}\}$

G is called ***factored*** *if all rules from R are factored.*

Tree adjoining grammars (TAGs) are based on tree adjunction. Here we shall give a formulation in terms of string adjunction. This allows us to directly compare the two approaches on the same class of languages, namely string languages. The idea is that the string language is the result of applying a certain homomorphism. We shall informally explain the idea. First we define **tree codings**. I shall use the notation $[\cdots]_C$ to denote a tree with root node labeled C. The $\cdots$ will be filled with a sequence of trees in the order they appear in the (ordered) tree. Strings are considered to be trees of height 0. For a string $\vec{x}$ put

(8) $\gamma(\vec{x}) := \vec{x}$

If $[\vec{x}]_A$ is a tree of height 1, then

(9) $\gamma([\vec{x}]_A) := (A\vec{x}A)$

In general, for $[C_1\ C_2\ \ldots\ C_n]_A$, where C_i are trees of arbitrary height, put

(10) $\gamma([C_1\ C_2\ \ldots\ C_n]_A) := (A\gamma(C_1)\gamma(C_2)\ldots\gamma(C_n)A)$

We formulate tree adjunction in terms of string adjunction on the representing string; there are however restrictions on how the rules and the adjunction string can look like. We start with unregulated adjunction. An adjunction string must have the form $\langle\vec{u},\vec{v}\rangle$ such that for any $\vec{y}$ that codes a tree of category X, $\vec{u}\vec{y}\vec{v}$ also codes a tree of category X. The locale of this rule is of the form

(11) $$\Lambda_X := \{\langle\vec{x}, (X\vec{y}X), \vec{z}\rangle : \vec{x},\vec{y},\vec{z} \in A^*\}$$

This finishes the definition of **unregulated** TAGs. Unregulated TAGs are factored contextual grammars. The string language they generate is obtained by applying the following homomorphism:

(12) $$h(a) := \begin{cases} \varepsilon & \text{if } a = (,) \text{ or } a \text{ a category symbol} \\ a & \text{else} \end{cases}$$

Regulated TAGs allow to additionally specify for each node which trees may be adjoined to it. It is known that modulo innocuous modification, regulated TAGs differ from unregulated TAGs only in that one may never adjoin twice to the same node. (See [Kracht, 2003] for a proof. The transformation preserves the strings we can generate (after applying h), so that we can always assume we have grammars of this form.) To this end we introduce a symbol $\lightning$ in the following way: the adjunction strings have the form

(13) $$\langle \lightning\vec{u}\lightning, \lightning\vec{v}\lightning\rangle$$

where $\langle\vec{u},\vec{v}\rangle$ is an admissible adjunction string for unregulated grammars. The locales are contained in

(14) $$\Lambda = \{\langle\vec{x}c, (a\vec{y}a), c'\vec{z}\rangle : c, c' \neq \lightning, a \in A\}$$

Here, a can only be a nonterminal. This says that adjunction cannot occur if the constituent is flanked by $\lightning$. This in turn can only arise if it has been formed by adjunction. The string language is obtained by applying

(15) $$h^{\lightning}(a) := \begin{cases} \varepsilon & \text{if } a = (,), \lightning \text{ or } a \text{ a category symbol} \\ a & \text{else} \end{cases}$$

Notice that TAGs are formulated using string adjunction, so they produce strings, not trees, the only difference being that these strings are codes for trees. However, the notion of a tree adjoining language is nevertheless a different one because the symbols used for coding the trees must be discarded.

Definition 5 *A string language L is a **tree adjoining language** if there is a TAG G such that $L = h^{\natural}[L(G)]$.*

Notice that TAGs also use an operation of *substitution*. Substitution is the result of adding a tree under a single nonterminal node. Substitution trees are not subject to any restriction, but the label of the root must match that of x in order to be substitutable there. Effectively, substitution can be subsumed under adjunction, if some structural modifications are being made. We shall however not discuss substitution any further.

4 Grammars for Interpreted Languages

In general, a grammar consists in a finite set of rules; a rule spells out a whole as a complex of parts. Seeing this bottom up we can specify the rule as a function that takes a complex and returns a whole. The complex is then simply a list of signs. Thus, we arrive at a notion of grammar that consists in a finite set of functions from signs to signs. So, f maps an n-tuple of signs to a sign.

(16) $\langle \sigma_0, \dots, \sigma_{n-1} \rangle \mapsto f(\sigma_0, \dots, \sigma_{n-1})$

The functions may be partial and they may even be indeterminate, that is, yield multiple outputs. On the syntactic side we shall require that these functions operate of the exponents irrespective of the meaning (this is certainly true for all brands of adjunction grammars); if in addition the grammar is *compositional* it also means that the functions operate on the meanings irrespective of the exponents.

Definition 6 (Compositional Grammar) *An n-**ary compositional rule** is a pair $f = \langle f^{\varepsilon}, f^{\mu} \rangle$ such that for $\sigma_i = \langle \vec{x}_i, m_i \rangle$, $i < n$, we have*

(17) $f(\sigma_0, \dots, \sigma_{n-1}) = \langle f^{\varepsilon}(\vec{x}_0, \dots, \vec{x}_{n-1}), f^{\mu}(m_0, \dots, m_{n-1}) \rangle$

*A **compositional grammar** is a finite set F of compositional rules.*

We actually allow the functions to be indeterminate, that is, to yield several values. In that case, however, if a syntactic function is indeterminate, the meanings should not depend on the particular choice. That is to say: if f^{ε} yields values $\vec{y}_{\bullet}$ and $\vec{y}_{\circ}$ on the input $\langle \vec{x}_0, \dots, \vec{x}_{n-1} \rangle$, and if f^{μ} gives values $n_{\bullet}$ and $n_{\circ}$ on the input $\langle m_0, \dots, m_{n-1} \rangle$, then all combinations, $\langle \vec{y}_{\bullet}, n_{\bullet} \rangle$, $\langle \vec{y}_{\bullet}, n_{\circ} \rangle$, $\langle \vec{y}_{\circ}, n_{\bullet} \rangle$, and $\langle \vec{y}_{\circ}, n_{\circ} \rangle$ are values of f of $\langle \sigma_0, \dots, \sigma_{n-1} \rangle$. Likewise, if any of f^{μ}, f^{ε} is undefined, so is f.

Finally we come to the notion of a compositional adjunction grammar. As we have indicated above, the syntactic operations are defined by a pair, giving a locale and an adjunction pair. The locale is checking for possible adjunction sites, and the adjunction string is what is inserted once a site has been found. These rules are (or may be) indeterminate. They are however always unary.

Definition 7 (Compositional SAG) *An **interpreted string adjunction rule** is a pair $h = \langle \rho, f \rangle$, where ρ is a string adjunction rule and f a relation between meanings. We put*

(18) $\langle \vec{x}, m \rangle \mapsto_h \langle \vec{y}, n \rangle$

*iff $\vec{x} \to_\rho \vec{y}$ and $m \to_f n$. A **compositional string adjunction grammar** is a pair $\langle U, F \rangle$, where U is a finite set of signs and F a finite set of interpreted string adjunction rules.*

We note the following. Say that L is **unambiguous** if $\langle \vec{x}, m \rangle, \langle \vec{x}, n \rangle \in L$ implies $m = n$. If L is unambiguous, then both f^ε and f^μ in the interpreted adjunction rule can be assumed to be partial functions. The languages we consider here are unambiguous, so we may from now on talk about (partial) functions rather than relations. In this paper we shall not deal with the case where the semantic functions are partial. This is because we are exploring the possibility of the syntax doing a proper job. The situation where the semantic functions are partial is far more complex and will be left out of consideration.

5 Some Basic Results

Let L be an interpreted language. Say that L has **finite semantics** if the number of possible semantic values is bounded. The first result is actually of positive nature. It establishes what *can* be done, given the full power of adjunction. We shall later see that TAGs are far weaker than this.

Theorem 8 *Let L be an interpreted language with finite semantics. If $s[L]$ can be generated by a string adjunction grammar, then there is an interpreted string adjunction grammar for L.*

Proof. Let $G = \langle B, R \rangle$ be a string adjunction grammar that generates $s[L]$. Let $S = \{s_0, \cdots, s_{n-1}\}$ be the set of semantic values. Put

(19) $B^+ := (B \times S) \cap L$

This set is finite. For each rule $\rho = \langle \alpha_\rho, \Lambda_\rho \rangle \in R$ with $\alpha = \langle \vec{u}, \vec{v} \rangle$ and each $i < n$ put

(20) $\Lambda_{\rho,i} := \{\langle \vec{x}, \vec{y}, \vec{z} \rangle \in \Lambda_\rho : \langle \vec{x}\vec{u}\vec{y}\vec{v}\vec{z}, s_i \rangle \in L\}$

Put $\rho_i := \langle \alpha_\rho, \Lambda_{\rho,i} \rangle$. For each i let $h_i(s) := s_i$. The set of interpreted rules is now

(21) $R^+ := \{\langle \rho_i, h_i \rangle : \rho \in G, i < n\}$

This defines the new grammar $G^+ := \langle B^+, R^+ \rangle$. We need to show that G^+ generates L. It is clear that the string set it generates equals $s[L]$, since $\Lambda_\rho = \bigcup_{i<n} \Lambda_{\rho,i}$. So we only need to show that the strings are associated with the right meanings. Suppose for this purpose that $\langle \vec{w}, s_i \rangle \in L$. Then there is a derivation of $\vec{w}$ in G. If the derivation has length 0, $\vec{w} \in B$. Hence $\langle \vec{x}, s_i \rangle \in B^+$. Now let the length be > 0. We look at the last step of the derivation. It consists in applying the rule ρ to a string $\vec{w}_1$. By definition there is a j such that $\langle \vec{w}_1, s_j \rangle \in L$. Now, again by definition, the rule ρ_i can be applied to this pair, and it yields $\langle \vec{w}, s_i \rangle$, as required. Notice also that we cannot derive $\langle \vec{w}, s_k \rangle$ unless this pair is in L, by definition of the rules. □

Notice that this proof does not work if we require the rules to be factored. The condition that L has finite semantics is necessary. To see this, let us take the following example.

Example 1. Let $\ell : \mathbb{N} \to \mathbb{N}$ be the following function: if n has the form 2^k then $\ell(n) := k$. Otherwise, $\ell(n) := 1$. Put

(22) $L := \{\langle \mathsf{a}^n, \ell(n) \rangle : n \in \mathbb{N}\}$

L has no adjunction grammar of any kind. To see this, notice that the rules consist in adding a pair of strings at some points in the string. Let p be the maximum of the combined lengths of the added strings. Then if a^n is produced from a^m in 1 step, then $n \leq m + p$. We pick $n = 2^k$ and look at the last step that produces the string. There is an adjunction step from a^m that delivers a^n. If k is large enough, m is not of the form 2^k. Thus we have the sign $\langle \mathsf{a}^m, 1 \rangle$. There are finitely many semantic functions, one for each rule. These are all functions from $\mathbb{N}$ to $\mathbb{N}$. Pick a function $f \in F$; it is unary, by definition of TAGs. Thus, from the sign $\langle \mathsf{a}^m, 1 \rangle$ we can create the sign $\langle \mathsf{a}^n, \ell(n) \rangle = \langle f^\varepsilon(\mathsf{a}^m), f^\mu(1) \rangle$. As F is finite, there are only finitely many values $f^\mu(1)$, but $\ell(n)$ can be anything. Contradiction. ♠

This example can be refined somewhat. We can show two things: first, that there exist genuine interpreted languages that can only be generated through adjunction. And second, that there exist interpreted CFLs whose sentences cannot be derived by an interpreted adjunction grammar. Let's start with the first example.

Example 2. Let $b(\vec{x})$ be the number of occurrences of the letter b in $\vec{x}$, and $a(\vec{x})$ the number of occurrences of the letter a.

(23) $$L := \{\langle \vec{x}\mathsf{c}\vec{x}^T, \ell^{b(\vec{x})}(a(\vec{x}))\rangle : \vec{x} \in (\mathsf{a} \mid \mathsf{b})^*\}$$

This has an interpreted adjunction grammar. We define first the string rules.

(24) $$\rho_0 := \langle\langle \mathsf{a}, \mathsf{a}\rangle, \{\langle \varepsilon, \mathsf{a}^n\mathsf{c}\mathsf{a}^n, \varepsilon\rangle : n \in \mathbb{N}\}\rangle$$

(25) $$\rho_1 := \langle\langle \mathsf{b}, \mathsf{b}\rangle, \{\langle \vec{x}, \vec{y}, \vec{x}^T\rangle : \vec{x}, \vec{y} \in (\mathsf{a} \mid \mathsf{b})^*\}\rangle$$

The base is c, from which everything is derived. Using ρ_0, we derive the strings $\mathsf{a}^n\mathsf{c}\mathsf{a}^n$. Once we apply ρ_1, ρ_0 can no longer be applied. ρ_1 inserts b into both strings, at mirror places. The semantic functions accompanying these rules are

(26) $$f_0(n) := n + 1$$

(27) $$f_1(n) := \ell(n)$$

The base is $\langle \mathsf{c}, 0\rangle$. Applying ρ_0 n times we get $\langle \mathsf{a}^n\mathsf{c}\mathsf{a}^n, n\rangle$. Applying ρ_1 after that we get the language L.

This language has no compositional CFG. For suppose G is such a grammar. Assume that it has k rules. Let $\tau(p, n)$ be the following sequence: $\tau(p, 0) := p$, $\tau(p, n + 1) := 2^{\tau(p,n)}$. Then $\ell(\tau(p, n + 1)) = \tau(p, n)$ for every n. The language contains the signs

(28) $$\sigma_{p,n} := \langle \mathsf{a}^{\tau(p,n)}\mathsf{b}^n\mathsf{c}\mathsf{b}^n\mathsf{a}^{\tau(p,n)}, p\rangle$$

Consider the last step generating $\sigma_{p,n}$. It must be formed from already existing signs of L. The string function is concatenation. Since each sign has a string containing c exactly once, $\sigma_{p,n}$ is formed from a single sign $\sigma = \langle \vec{x}, m\rangle$ adding strings to the left and right of $\vec{x}$. It is easy to see that for large enough n, the strings added consist entirely of as. So, we have a unary function symbol f such that

(29) $$\sigma_{p,n} = \langle f^\varepsilon(\vec{x}), f^\mu(m)\rangle$$

and $f^\varepsilon(\vec{x}) = \mathsf{a}^k\vec{x}\mathsf{a}^k$ for some k. Thus, provided that n is large enough

(30) $$\sigma = \langle \mathsf{a}^{\tau(p,n)-k}\mathsf{b}^n\mathsf{c}\mathsf{b}^n\mathsf{a}^{\tau(p,n)-k}, 1\rangle$$

So, $f^\varepsilon(1) = p$. However, p was arbitrarily chosen and we have only k many rules. Contradiction. ♠

Notice that this example works also if we restrict the language to strings that contain only one pair of bs.

Example 3.

(31) $$M := \begin{array}{ll} & \{\langle \mathsf{a}^m, m\rangle : m \in \mathbb{N} - \{0\}\} \\ \cup & \{\langle \mathsf{b}^m, m\rangle : m \in \mathbb{N} - \{0\}\} \\ \cup & \{\langle \mathsf{a}^m\mathsf{b}^n, \ell(m) + \ell(n)\rangle : m, n \in \mathbb{N} - \{0\}\} \end{array}$$

Here is a CFG. The rules are

(32) $$\mathtt{S} \to \mathtt{AB} \qquad \mathtt{A} \to \mathtt{aA} \mid \mathtt{a} \qquad \mathtt{B} \to \mathtt{bB} \mid \mathtt{b}$$

This can be turned into an interpreted CFG as follows. There are five interpreted rules. There are two zeroary rules: $\rho_0 := \langle \mathsf{a}, 1\rangle$, $\rho_1 := \langle \mathsf{b}, 1\rangle$.

(33) $$\rho_2(\langle \vec{x}, m\rangle) := \begin{cases} \langle \mathsf{a}\vec{x}, m+1\rangle & \text{if } \vec{x} \in \mathsf{a}^* \\ \text{undefined} & \text{else} \end{cases}$$

(34) $$\rho_3(\langle \vec{x}, m\rangle) := \begin{cases} \langle \mathsf{b}\vec{x}, m+1\rangle & \text{if } \vec{x} \in \mathsf{b}^* \\ \text{undefined} & \text{else} \end{cases}$$

(35) $$\rho_4(\langle \vec{x}, m\rangle, \langle \vec{y}, n\rangle) := \begin{cases} \langle \vec{x}\vec{y}, \ell(m) + \ell(n)\rangle & \text{if } \vec{x} \in \mathsf{a}^*, \vec{y} \in \mathsf{b}^* \\ \text{undefined} & \text{else} \end{cases}$$

The partial string functions implicitly define the categories. To see that there is no adjunction grammar, we appeal to the fact that the rules are unary. Let G be such a grammar, and let it have k rules. Choose s and t arbitrarily and let $m = 2^s$ and $n = 2^t$. For large enough s and t we cannot derive $\mathsf{a}^m\mathsf{b}^n$ by concatenating a^m and b^n. Rather, we have to derive it from a string of the form $\mathsf{a}^p\mathsf{b}^q$ where there is a k (depending only on the grammar) such that $m < p + k$ and $n < q + k$. Also, there are a finite number of rules. Now, suppose the string has been derived from $\mathsf{a}^p\mathsf{b}^q$. We assume first that no adjunction rule inserts only as or only bs. The input sign is then

(36) $$\langle \mathsf{a}^p\mathsf{b}^q, \ell(p) + \ell(q)\rangle$$

where $p < m$ and $q < n$. What can be shown is that there is some number—namely 2—from which an unbounded number of different output meanings must be computed. Then $\ell(p) + \ell(q) = 2$, but $\ell(m) + \ell(n) = s + t$. But s and t were arbitrary. So, for every large enough number $s + t$ we must have an f such that $f^{\varepsilon}(1) = s + t$. But we have only k many functions. Contradiction. The case where some rule inserts only as or only bs is somewhat more complex but uses the same method. ♠

What these examples show is that the question whether a language has a compositional grammar clearly depends on the type of string functions one is prepared to admit. Thus it is not possible to say that adjunction grammars are incorrect or that context free grammars are incorrect. Thus the question rather is which of these grammar types is suited for the languages that we are interested in. These are, of course, natural languages, programming languages and formal languages.

6 Interpreted Boolean Expressions

To assess the question of the usefulness of adjunction we turn to a rather simple language, the language of boolean expressions. The alphabet consists of the symbols

(37) $A = \{\mathtt{p}, \mathtt{(}, \mathtt{)}, \mathtt{0}, \mathtt{1}, \neg, \wedge, \vee\}$

An **index** is a sequence of $\mathtt{0}$ and $\mathtt{1}$. A **variable** is a string of the form $\mathtt{p}\vec{x}$, where $\vec{x}$ is an index. $\vec{z}$ is a formula iff either

1. $\vec{z}$ is a variable or
2. $\vec{z} = \mathtt{(}\vec{x}\wedge\vec{y}\mathtt{)}$, where $\vec{x}$ and $\vec{y}$ are formulae, or
3. $\vec{z} = \mathtt{(}\vec{x}\vee\vec{y}\mathtt{)}$, where $\vec{x}$ and $\vec{y}$ are formulae, or
4. $\vec{z} = \mathtt{(}\neg\vec{x}\mathtt{)}$, where $\vec{x}$ is a formula.

The set of formulae is denoted by Form. This language is fairly simple. It is a (somewhat unusual) version of boolean logic. The quirk we have added (to simplify matters a little bit) is to enforce strict bracketing. This is a needless but useful restriction. We deal with this language both in terms of string adjunction and in terms of tree adjunction. When talking about tree adjunction we suppress however the additional tree coding (consisting of the nonterminal and the adjunction prohibition), except for Example 4.

Definition 9 *A **valuation** β is a function from the set of indices into $2 = \{1, 0\}$. V is the set of all valuations. The **meaning** $[\vec{x}]$ of a formula $\vec{x}$ is defined as follows.*

1. *If $\vec{x} = \mathtt{p}\vec{y}$: $[\vec{x}] := \{\beta : \beta(\vec{y}) = 1\}$.*
2. *If $\vec{x} = \mathtt{(}\neg\vec{y}\mathtt{)}$: $[\vec{x}] := V - [\vec{y}]$.*

3. *If* $\vec{x} = (\vec{y}\wedge\vec{z})$*:* $[\vec{x}] := [\vec{y}] \cap [\vec{z}]$.

4. *If* $\vec{x} = (\vec{y}\vee\vec{z})$*:* $[\vec{x}] := [\vec{y}] \cup [\vec{z}]$.

The interpreted language is

(38) $$\mathrm{Bool} := \{\langle \vec{x}, [\vec{x}]\rangle : \vec{x} \in \mathrm{Form}\}$$

It is easy to see that Bool has a compositional CFG.[4] This grammar knows one constant: $\rho_0 := \langle \varepsilon, \varepsilon\rangle$. Furthermore, it has two unary rules ρ_1 and ρ_2 to construct indices and ρ_3 to make variables. An *index* is a string free of occurrences of $\mathtt{p}$.

(39) $$\rho_1(\langle \vec{x}, \vec{y}\rangle) := \begin{cases} \langle \vec{x}^\frown \mathtt{0}, \vec{y}^\frown \mathtt{0}\rangle & \text{if } \vec{x} \text{ is an index} \\ \text{undefined} & \text{else} \end{cases}$$

(40) $$\rho_2(\langle \vec{x}, \vec{y}\rangle) := \begin{cases} \langle \vec{x}^\frown \mathtt{1}, \vec{y}^\frown \mathtt{1}\rangle & \text{if } \vec{x} \text{ is an index} \\ \text{undefined} & \text{else} \end{cases}$$

(41) $$\rho_3(\langle \vec{x}, \vec{y}\rangle) := \begin{cases} \langle \mathtt{p}^\frown \vec{x}, [\vec{y}]\rangle & \text{if } \vec{x} \text{ is an index} \\ \text{undefined} & \text{else} \end{cases}$$

(Notice that one can only generate pairs of the form $\langle \vec{x}, \vec{y}\rangle$ where $\vec{x} = \vec{y}$, so the rules appear to be more general than they actually are.) Similarly, there is one unary function to create negated expressions and two binary functions for disjunction and conjunction, respectively.

(42) $$\rho_4(\langle \vec{x}, m\rangle) := \begin{cases} \langle \mathtt{(}^\frown \neg^\frown \vec{x}^\frown \mathtt{)}, V - m\rangle & \text{if } \vec{x} \text{ is not an index} \\ \text{undefined} & \text{else} \end{cases}$$

(43) $$\rho_5(\langle \vec{x}, m\rangle, \langle \vec{y}, n\rangle) := \begin{cases} \langle \mathtt{(}^\frown \vec{x}^\frown \wedge^\frown \vec{y}^\frown \mathtt{)}, m \cap n\rangle & \text{if } \vec{x}, \vec{y} \text{ are not indices} \\ \text{undefined} & \text{else} \end{cases}$$

(44) $$\rho_6(\langle \vec{x}, m\rangle, \langle \vec{y}, n\rangle) := \begin{cases} \langle \mathtt{(}^\frown \vec{x}^\frown \vee^\frown \vec{y}^\frown \mathtt{)}, m \cup n\rangle & \text{if } \vec{x}, \vec{y} \text{ are not indices} \\ \text{undefined} & \text{else} \end{cases}$$

[4]This grammar produces an additional kind of string—the index—, which is not standardly assumed in the syntax of propositional logic for the reason that the propositional letters are atomic. It is possible (at the expense of using nonstandard semantic functions) to avoid postulating constituents of type index, as we show below. However, it is more natural to use the grammar shown here.

Notice that it is not necessary to explicitly code the category: there are two types of strings, indices and other strings, and indices are easy to recognize.

The question is whether this language has a compositional adjunction grammar. The simplest case we can imagine is the sublanguage of all variables, that is, the set

(45) $\mathrm{Var} := \{\langle \mathrm{p}\vec{x}, [\mathrm{p}\vec{x}]\rangle : \vec{x} \text{ an index}\}$

This language already raises difficulties for unregulated adjunction.

Theorem 10 *There is no compositional unregulated TAG that generates* Var.

Proof. Let G be an unregulated TAG based on p adjunction trees. We shall find a string such that there are more than p sites where a tree can adjoin (no matter what analysis is given), and that adjunction will lead to a different string in each case. To this end let f be the longest yield of a tree in G. (Recall that the yield of a tree is the string defined by the concatenation of its leaves.) For n large, pick the string consisting of $2p + 3$ repetitions of $1^n \mathtt{0}^{2f+1}$. For each of these blocks, one continuous stretch of zeros has been introduced through adjunction of some tree. Adjoining this tree again will increase the length of the continuous stretch of zeros in this block. It may also add some other string elsewhere. We have in total $2p + 3$ places where we can increase the stretch of zeros. It is not hard to see that at least $p + 1$ such adjunctions must yield a different result. Contradiction. □

The problem is that variables are strings of unbounded length and therefore have unboundedly many adjunction sites. TAGs by contrast allow to use adjunction prohibition and thereby allow to control the number of adjunction sites.

Example 4. Here is a TAG for Var. Strings are analysed as left branching, as if they were bracketed as follows.

(46) `(((((p)1)1)0)0)`

The adjunction pairs are

(47) $\langle \text{↯}\mathtt{(S(S}\text{↯}\mathtt{,}\text{↯}\mathtt{0S)S)}\text{↯}\rangle$, $\langle \text{↯}\mathtt{(S(S}\text{↯}\mathtt{,}\text{↯}\mathtt{1S)S)}\text{↯}\rangle$

The base string is

(48) `(SpS)`

The derivation for (46) is as follows:

(49)
$$\begin{array}{l}
\mathtt{(SpS)} \\
\underline{\mathtt{\lightning(S(S\lightning}}\mathtt{(SpS)}\underline{\mathtt{\lightning 1S)S)\lightning}} \\
\mathtt{\lightning(S}\underline{\mathtt{\lightning(S(S\lightning}}\mathtt{(S\lightning(SpS)\lightning 1S)}\underline{\mathtt{\lightning 1S)S)\lightning}}\mathtt{S)\lightning} \\
\mathtt{\lightning(S\lightning(S}\underline{\mathtt{\lightning(S(S\lightning}}\mathtt{(S\lightning(S\lightning(SpS)\lightning 1S)\lightning 1S)}\underline{\mathtt{\lightning 0S)S)\lightning}}\mathtt{S)\lightning S)\lightning} \\
\mathtt{\lightning(S\lightning(S\lightning(S}\underline{\mathtt{\lightning(S(S\lightning}}\mathtt{(S\lightning(S\lightning(S\lightning(SpS)\lightning 1S)\lightning 1S)\lightning 0S)}\underline{\mathtt{\lightning 1S)S)\lightning}}\mathtt{S)\lightning S)} \\
\qquad \mathtt{\lightning S)\lightning}
\end{array}$$

Apply the deletion homomorphism: $\mathtt{(}$, $\mathtt{)}$, $\mathtt{S}$ and $\lightning$ are all mapped to ε. This gives us (46). ♠

Let us also define the following language. Let β be a valuation. Then let

(50) $$\mathrm{Var}^\beta := \langle \mathtt{p}\vec{x}, \beta(\vec{x})\rangle$$

Then the languages Var^β have finite semantics and the string language can be generated by an unregulated adjunction grammar. Nevertheless, for certain choices of β there exists no unregulated interpreted TAG for these languages. (For example, choose β such that $\{\vec{x} : \beta(\vec{x}) = 1\}$ is not recursive.) Contrast this with Theorem 8 which states that there is an adjunction grammar for this language. This shows that adjunction grammars can generate interpreted languages for which there is even no interpreted CFG.

7 There Are No TAGs for Boolean Expressions

In oder to show that a language has no grammar of a certain kind it is enough to study a fragment in the following sense.

Definition 11 *Let $L \subseteq A^* \times M$ be an interpreted language and $B \subseteq A$. Then $L \upharpoonright B := L \cap (B^* \times M)$ is the B-**fragment** of L.*

For example, Var is the $\{\mathtt{0}, \mathtt{1}, \mathtt{p}\}$-fragment of Bool. In addition to Var we shall study the following fragments of Bool:

(51)
$$\begin{array}{l}
\mathrm{Bool}^{\wedge} := \mathrm{Bool} \upharpoonright \{\mathtt{0}, \mathtt{1}, \mathtt{p}, \wedge\} \\
\mathrm{Bool}^{\neg} := \mathrm{Bool} \upharpoonright \{\mathtt{0}, \mathtt{1}, \mathtt{p}, \neg\}
\end{array}$$

Now assume G is a compositional grammar for L. Then for every f, let

(52) $$f \upharpoonright B := \langle f^\varepsilon \upharpoonright B, f^\mu \rangle$$

Here, $f^\varepsilon \restriction B := f^\varepsilon \cap B^{n+1}$, which is to say that $(f^\varepsilon \restriction B)(\vec{x}_0, \cdots, \vec{x}_{n-1})$ is defined iff for all $i < n$ $\vec{x}_i \in B^*$ *and* if $f^\varepsilon(\vec{x}_1, \cdots, \vec{x}_n) \in B^*$. Now suppose further that our grammar is **additive**, by which we mean to say that for every $f \in G$ $f^\varepsilon(\vec{x}_0, \cdots, \vec{x}_{n-1})$ contains every letter of A at least as often as in the $\vec{x}_i$ together. Adjunction grammars certainly are additive. Then if all the $\vec{x}_i$ are in B^*, so is $f^\varepsilon(\vec{x}_0, \cdots, \vec{x}_{n-1})$. Hence we have a grammar

(53) $G \restriction B := \{f \restriction B : f \in G\}$

Now, $G \restriction B$ generates a subset of L, by construction. Moreover, it only creates signs from $L \restriction B$. In fact, it generates *exactly* $G \restriction B$. Namely, we show by induction on the length of $\vec{x}$ that if $\langle \vec{x}, m\rangle$ in $L \restriction B$ then $\langle \vec{x}, m\rangle \in L(G \restriction B)$. So assume that the claim holds for all strings shorter than $\vec{x}$. We have

(54) $\langle \vec{x}, m\rangle = \langle f^\varepsilon(\vec{x}_1, \cdots, \vec{x}_n), f^\mu(m_1, \cdots, m_n)\rangle$

for some n. By assumption, $f^\varepsilon(\vec{x}_1, \cdots, \vec{x}_n)$ contains every letter at least as often as the $\vec{x}_i$ together. So, if letters from $A - B$ do not occur in $\vec{x}$, they do not occur in any of the $\vec{x}_i$ either. So we have $\langle \vec{x}_i, m_i\rangle \in L \restriction B$. They all have shorter length, the induction hypothesis applies to them. It follows that

(55) $\langle \vec{x}, m\rangle = \langle (f^\varepsilon \restriction B)(\vec{x}_1, \cdots, \vec{x}_n), f^\mu(m_1, \cdots, m_n)\rangle$

Thus $\langle \vec{x}_i, m_i\rangle \in L(G \restriction B)$, showing $\langle \vec{x}, m\rangle \in L(G \restriction B)$. (There is just one exception to be handled, namely when $f^\varepsilon(\vec{x}_1, \cdots, \vec{x}_n)$ is identical to one of the $\vec{x}_i$. In that case we do a subinduction on the number of steps.)

Proposition 12 *Suppose that G is an additive compositional grammar for L. Then $G \restriction B$ is an additive compositional grammar for $L \restriction B$.*

Thus if G is an adjunction grammar (unregulated TAG, TAG) so is $G \restriction B$.

Call a **literal** a formula that contains no operation symbols other than $\neg$; moreover, $\neg$ may occur at most once. Call a **clause** a formula made from literals using only $\wedge$. Put

(56) $\mathrm{Cls} := \{\langle \vec{x}, [\vec{x}]\rangle : \vec{x} \text{ is a clause}\}$

Let us see how one might define a TAG for Cls. We start by giving a TAG for the sublanguage Lit.

(57) $\mathrm{Lit} := \{\langle \vec{x}, [\vec{x}]\rangle : \vec{x} \text{ is a literal}\}$

Lit can be generated in the following way. The base is given by:

(58) $B = \{\langle \mathtt{p}, [\mathtt{p}]\rangle, \langle \mathtt{(\neg p)}, [\mathtt{(\neg p)}]\rangle\}$

The adjunction operations are given by the adjunction strings $\alpha_0 = \langle \mathtt{0}, \varepsilon\rangle$ and $\alpha_1 = \langle \mathtt{1}, \varepsilon\rangle$, both operating using the locale

(59) $\Lambda = \{\langle \vec{x}, \varepsilon, \vec{y}\rangle : \vec{x} \in \{\mathtt{p}, \mathtt{(\neg p}\}, \vec{y} \in A^*\}$

(60) $\rho_0 = \langle \alpha_0, \Lambda\rangle, \qquad \rho_1 = \langle \alpha_1, \Lambda\rangle$

Thus, the operations will insert either a $\mathtt{0}$ or a $\mathtt{1}$ right after the $\mathtt{p}$. Operations that add only $\mathtt{0}$ or $\mathtt{1}$ shall be called **index shifts**.

The map $\vec{y} \mapsto [\mathtt{p}\vec{y}]$ is injective and has an inverse, which we denote by †. Let

(61) $$\begin{aligned} g_0(U) &:= [\mathtt{p0}(U^\dagger)] \\ g_1(U) &:= [\mathtt{p1}(U^\dagger)] \end{aligned}$$

Given a set of the form $[\mathtt{p}\vec{y}]$, we have

(62) $g_0([\mathtt{p}\vec{y}]) = [\mathtt{p0}([\mathtt{p}\vec{y}]^\dagger)] = [\mathtt{p0}\vec{y}]$

The grammar is $\langle B, \{\langle \rho_0, g_0\rangle, \langle \rho_1, g_1\rangle\}\rangle$. It generates the expressions by mirroring the string substitution by an exchange in the variables to which the valuations effectively respond. Namely, the sets are always of the form the value of a particular variable is 0 (or 1). That special variable is exchanged when the index changes. This example can be upgraded to a grammar for $\mathrm{Bool}^\neg$. What we need in addition is the adjunction rule $\rho_2 := \langle\langle \mathtt{(\neg(\neg}, \mathtt{))}\rangle, \Lambda'\rangle$.

(63) $\Lambda' := \{\varepsilon\} \times A^* \times \{\varepsilon\}$

The accompanying semantic function is the identity $i : m \mapsto m$.

Theorem 13 *$\langle B, \{\langle \rho_0, g_0\rangle, \langle \rho_1, g_1\rangle, \langle \rho_2, i\rangle\}\rangle$ is a compositional TAG for* $\mathrm{Bool}^\neg$.

Below we consider the fragment $\mathrm{Bool}^\wedge$. Here we consider briefly the language of clauses. A clause is said to **contain** a given literal ℓ if ℓ is a substring of the clause and not in the scope of $\neg$. (So, $\mathtt{p01}$ is *not* contained in $\mathtt{(p\wedge(\neg p01))}$, but $\mathtt{(\neg p01)}$ is.) First of all notice that $[\vec{x}] = \varnothing$ if $\vec{x}$ contains both a literal and its negation. It is this case that needs special attention. Notice that while the mapping from syntax to semantics is unique, a given semantic value has infinitely many strings that have it as their meaning. This is because we cannot tell whether a given literal occurs once or twice or three times; we can at best determine whether it does or does not occur. Second, the order of the literals cannot be recovered. And third the associative structure cannot be recovered. So, there are two cases. Case 1. The clause is consistent (its set of satisfying valuations is not empty). Then we can determine exactly what literals occur in it. Case 2. The clause is not consistent. Then we only know that it contains *some literal and its negation*. That's all.

To make this more readily visible, we decide to represent the sets $[\vec{x}]$ in the following way. A **p-set** is a pair (U, U') of sets of indices such that either (1) $U \cap U' = \varnothing$ or (2) $U = U' =$ the set of all indices. The last value is denoted by $\star$. If (1) obtains, the clause is consistent, and the set U indicates on which variables the valuation must give 1; while the set U' indicates on which variables the valuation must give 0. If (2) obtains, no valuation exists.

Consider as before the operation that adds `0` right after an occurrence of `p`. If nothing else is known, the output for the semantics is indeterminate. Let us see why. In a clause of the form `((p0∧p1)∧(¬p01))` the index shift ρ_0 can in principle apply to three occurrences of variables. (Potentially, we could also insert several digits, and at other places. Nothing of substance would change, though.) The results are:

```
        ((p00∧p1)∧(¬p01))
(64)    ((p0∧p01)∧(¬p01))
        ((p0∧p1)∧(¬p001))
```

The result of an index shift on the p-set (U, U') depends on two additional conditions: we have to choose which index is targeted, and second, whether the index that is targeted belongs to a literal that occurs only once, or whether it belongs to a literal that occurs twice. For consider that we target the index $\vec{x}$ and that $\vec{x}$ occurs only once, say in U. Then the new p-set is $((U - \{\vec{x}\}) \cup \{\mathtt{0}\vec{x}\}, U')$. (This holds if these sets are disjoint—if they are not, the result is $\star$.) If however the literal occurs twice, then the result is rather $(U \cup \{\mathtt{0}\vec{x}\}, U')$, since only one occurrence of $\vec{x}$ has been touched. Notice that we do *not* need to specify whether the index is in U or in U', since the two sets are disjoint.

Thus, given a p-set (U, U') there are potentially as many outcomes as there are members of U and U' together. Since the grammar will have only a bounded number of functions, this is a point where one will have to restrict the adjunction sites via the locale.

A last case needs to be looked at. This is the case $U = U' = V$. This case means that there is a variable which occurs once plain and once negated. The semantics does not give away which one it is nor how many times it occurs. Thus, the result of an index shift can be almost anything. Again, given that the grammar has only finitely many functions, the result of applying any of these functions to (V, V) is any one of a fixed set H of p-sets. This in turn means that adjunction to an inconsistent clause is severely limited, because it may not target any variable which will remove the inconsistency unless the resulting clause belongs to one specified by H.

We shall now enter the proof that there is no TAG for the conjunctive fragment, It will proceed under the assumption that the semantic functions are total. So the only source of partiality is the ban on adjunction. However, this ban is a force to be reckoned with. Consider a grammar G and a big tree. If there is no ban on adjunction it is clear that the number of adjunction sites must grow with the length of the tree. In a TAG, however, adjunction sites can be closed in the course of a derivation. Consider now how adjunction can be blocked. [5] Each adjunction adds new nodes, to which one can potentially adjoin. Let $n(T)$ be the number of nodes to which one can adjoin, the **adjunction balance**. This is defined for center trees and adjunction trees. If T is an adjunction tree with $n(T) = 1$ the number of adjunction sites stays constant when adjoining T. (Each adjunction removes one site.) Call T a **plug** if $n(T) = 0$. Adjoining a plug decreases the number of adjunction sites. Suppose we have a tree T_0, we adjoin a plug T at a node x, and then adjoin to some node y some tree T'. Then $y \neq x$, since adjunction prohibits any further adjunction. Moreover, as T is a plug, y is not contained in T. So, y is a node of T_0, and we can actually adjoin T' to y before adjoining T to x, with the same resulting tree.

Lemma 14 *In a TAG, adjunction of a plug commutes with all other adjunctions.*

It follows that if there is a derivation for some tree T, there is a derivation where all plugs are added at the end. Moreover, the order in which the plugs are added can be chosen arbitrarily.

Next we look in detail at the possible adjunction strings for the language. For a string $\vec{x}$ an a symbol a, let $\sharp_a(\vec{x})$ denote the number of occurrences of a in $\vec{x}$. It is not difficult to see that in a string for Bool$^\wedge$

(65) $\sharp_{(}(\vec{x}) = \sharp_{)}(\vec{x}) = \sharp_{\wedge}(\vec{x}) = \sharp_{\mathrm{p}}(\vec{x}) - 1$

[5]The following passage appeals to our intuitions on trees. Of course we can use them despite the fact that the TAGs are formulated as string adjunction grammars. We shall switch between these viewpoints without warning.

Thus, for an adjunction string $\langle\vec{x},\vec{y}\rangle$ we must have

(66) $\quad \sharp_{\texttt{(}}(\vec{x}\vec{y}) = \sharp_{\texttt{)}}(\vec{x}\vec{y}) = \sharp_{\wedge}(\vec{x}\vec{y}) = \sharp_{\texttt{p}}(\vec{x}\vec{y})$

The number of `0`s and `1`s by contrast is unconstrained. For a given string call $\sharp_{\texttt{(}}(\vec{x}) - \sharp_{\texttt{)}}(\vec{x})$ the **balance** of $\vec{x}$. Let us say that $\vec{x}$ is **semibalanced** if the balance of every prefix of $\vec{x}$ is nonnegative. An adjunction string $\langle\vec{x},\vec{y}\rangle$ is semibalanced if $\vec{x}\vec{y}$ is.

Lemma 15 *Every adjunction string for* Bool *is semibalanced.*

Proof. Assume the contrary. Choose $\langle\vec{x},\vec{y}\rangle$ of minimal length such that it is not semibalanced and there is $\vec{r} = \vec{u}\vec{x}\vec{v}\vec{y}\vec{w} \in$ Bool as well as $\vec{u}\vec{v}\vec{w} \in$ Bool. Furthermore, assume $\vec{r}$ of minimal length satisfying the previous. If $\vec{x}\vec{y}$ contains an occurrence of `(` before an occurrence of `)` it must contain a subformula, and we may replace that subformula in $\vec{r}$ by the letter `p`. This will constitute a shorter counterexample in contradiction to our choice of $\langle\vec{x},\vec{y}\rangle$. So, we can assume that all the `(` follow all the `)`. Since $\vec{r} \in$ Bool it is itself semibalanced, and so the first occurrence of `)` in $\vec{x}\vec{y}$ must be matched by a preceding `(` in either $\vec{u}$ or $\vec{v}$. If another matching pair of brackets intervenes between the previous one, we can reduce it by `p`, to obtain a shorter example. So we can assume that the occurrence of `(` immediately precedes the designated occurrence of `)`. The newly added brackets are marked by underlining. No other brackets intervene.

(67) $\quad \cdots\texttt{(}\cdots\underline{\texttt{)}}\cdots\underline{\texttt{(}}\cdots\texttt{)}\cdots$

Then the original situation is this (fitting in arbitrary binary strings):

(68) $\quad \texttt{(p001}\wedge\texttt{p11)}$

The closing bracket must be inserted after the second occurrence of `p`.

(69) $\quad \texttt{(p001}\wedge\texttt{p1}\underline{\texttt{)}\wedge\texttt{(p01}\wedge\texttt{p}}\texttt{1)}$

Now, however, the outer brackets are missing. Adjunction however can only introduce one of them (for we must insert three discontinuous strings). □

The most powerful constraint comes from the following. Suppose you have a tree T that can be adjoined at two places. Since the two adjunctions are the same syntactic operation in a TAG, the resulting structures must have the same meaning (see Definition 7). This constrains the adjunctions severely. For example: let v_1 be a variable having occurrences $o_1, o_2, \ldots, o_n$ ($n > 2$) of the same variable v_1 such that the same tree can be adjoined to each of them, giving us an occurrence of a variable $v_2 \neq v_1$. After you have done one adjunction, v_2 will have an occurrence in the formula. Now you do the second adjunction: this removes the second occurrence of v_1, and adds another occurrence of v_2. The semantic function is identity. After you have done $n - 1$ adjunctions, removing all but the last occurrence, the final adjunction must however be accompanied by a semantic function that is not the identity: the resulting formula does not contain v_1 any more. It is however impossible to prevent the application of the rule that has the accompanying semantic function the identity. (This argument works irrespective of the partiality of the semantic functions.) Thus such a situation must be prevented.

We can get multiple occurrences as follows. Suppose there is an adjunction tree T (basic or derived) such that

① adjunction is licit on the root node of T,

② and T introduces an entire formula to which adjunction is possible.

If we can show the existence of a tree T and a formula whose derivation uses T, then we have the desired contradiction. Now, if T does *not* satisfy ① and ② then either

(a) T disallows adjunction to its root node, or

(b) T fails to introduce even a single complete formula into either part of the adjunction string, or

(c) whenever T chooses to introduce a complete formula in an adjunction string, that formula cannot be adjoined to.

Note that Option (c) means that the variables introduced by T cannot be adjoined to, that is, cannot grow in size.

Let G be a compositional TAG. We shall present some formulae such that a grammar generating them must have a tree satisfying ① and ②. The set of formulae is the following. Call a formula $\vec{x}$ n-**homogeneous of order** p if either $n = 0$ and $\vec{x}$ is a variable of length $> p$ or it is the conjunction of two $n - 1$-homogeneous formulae of order p. We show the following, which will establish our main result:

> Let G be a TAG for $\mathrm{Bool}^{\wedge}$. Then there is p and n and an n-homogeneous formula of order p which is not derivable in G without a tree satisfying ① and ②.

So, let us assume that $\vec{x}$ is n-homogeneous for some n and some order p. If n and p are sufficiently large, the derivation cannot use trees of Type (c). For once such a variable is introduced, it cannot grow in length.

Thus we can assume that all trees used in deriving $\vec{x}$ do not satisfy (c). We now analyse in some depth the possible adjunction strings. Notice that the adjunction strings must be semibalanced. Furthermore, we can restrict our attention to cases in which just one of each brackets is introduced. For either (i) one part of the adjunction string contains a sequence $\mathtt{(}\cdots\mathtt{)}$, and so a formula. Or (ii) the adjunction string has the form $\langle\cdots\mathtt{(}\cdots\mathtt{(}\cdots\mathtt{,}\cdots\mathtt{)}\cdots\mathtt{)}\ \cdots\rangle$. In Case (i), we first introduce the string without the subformula (just introducing $\mathtt{p}$) and require adjunction (by chosing an appropriate label) that will introduce the remainder. In Case (ii) we pretty much do the same: the innermost brackets enclose a formula, and so do the outermost brackets. We can perform the same adjunction in two stages.

Assume that this is so. This means that in underived adjunction strings, $\mathtt{(}$ is in the left part, $\mathtt{)}$ in the right part. We distinguish two subcases: (I) $\wedge$ is in the left part, (II) $\wedge$ is in the right part.

(Case I). $\wedge$ is in the left part. Then the adjunction pair looks like this (ignoring $\mathtt{0}$, $\mathtt{1}$ and $\mathtt{p}$):

(70) $\langle\cdots\wedge\cdots\mathtt{(}\cdots\mathtt{,}\cdots\mathtt{)}\cdots\rangle$

There can in fact be no symbol between $\wedge$ and $\mathtt{(}$ as no binary string can occur there. Furthermore, after a closing bracket a formula can contain only $\wedge$ or $\mathtt{)}$. This reduces the adjunction string to the following form:

(71) $\langle\cdots\wedge\mathtt{(}\cdots\mathtt{,}\cdots\mathtt{)}\rangle$

Since neither part of the adjunction string contains a complete formula, $\mathtt{p}$ must occur right after $\mathtt{(}$. (72) is the final form, where dots represent some binary string.

(72) $\langle\cdots\wedge\mathtt{(p}\cdots\mathtt{,}\cdots\mathtt{)}\rangle$

(Case II). $\wedge$ is in the right part. Since neither part contains an entire formula, $\wedge$ does not precede $\mathtt{)}$.

(73) $\langle\cdots\mathtt{(}\cdots\mathtt{,}\cdots\mathtt{)}\cdots\wedge\cdots\rangle$

$\mathtt{(}$ can only be preceded by $\mathtt{(}$ and $\wedge$ in a formula; thus $\mathtt{(}$ is not preceded by anything. Also, $\wedge$ must follow $\mathtt{)}$ immediately. So, we are left with the following choices, with dots representing binary strings:

(74) $\langle\mathtt{(p}\cdots\mathtt{,}\cdots\mathtt{)}\wedge\cdots\rangle$ $\qquad\langle\mathtt{(}\cdots\mathtt{,}\cdots\mathtt{)}\wedge\mathtt{p}\cdots\rangle$

$\wedge$ is not followed by a binary symbol, so we get

(75) $\quad (\alpha)\ \langle\texttt{(p}\cdots\texttt{,}\cdots\texttt{)}\wedge\rangle \qquad (\beta)\ \langle\texttt{(}\cdots\texttt{,}\cdots\texttt{)}\wedge\texttt{p}\cdots\rangle$

The type (α) is ruled out since it can nowhere be entered into a formula. To see this, let $\langle\vec{u}, \vec{v}, \vec{w}\rangle$ be the context. The result of adjunction is

(76) $\quad \vec{u}\texttt{(}\underline{\texttt{p}\cdots}\vec{v}\underline{\cdots\texttt{)}}\wedge\vec{w}$

What the brackets enclose in (76) is a formula. Also, $\vec{w}$ has a prefix $\vec{x}$ that is a formula. $\vec{v}$ may not begin with $\cdots\texttt{)}$ nor with $\cdots\texttt{(}$, so it begins with $\cdots\wedge$. And it ends likewise in $\texttt{p}\cdots$ or in $\texttt{)}$. Since $\vec{u}\vec{v}\vec{w}$ is a formula and contain the sequence $\vec{v}\vec{x}$, $\vec{x}$ cannot begin with a bracket, and so must be a variable. And $\vec{v}$ cannot end in $\texttt{)}$, neither can it end in $\texttt{(}\cdots$. Contradiction. Thus only the Type (β) needs to be considered. In this type, $\texttt{(}$ cannot be followed by a binary string. So we are down to the Case (77).

(77) $\quad \langle\texttt{(,}\cdots\texttt{)}\wedge\texttt{p}\cdots\rangle$

In Case (I) there is an occurrence of $\wedge$ that is not immediately preceded by a closing bracket, and in Case (II) there is an occurrence of $\wedge$ that is not followed by an occurrence of an opening bracket.

Now that we know about their identity, let us check where these adjunction strings can be inserted. In both cases it is easy to see that the kernel must contain an occurrence of $\wedge$. The occurrence of $\texttt{p}$ in (77) cannot be separated by an opening bracket from its preceding $\wedge$. For if we did this, we would have to insert a closing bracket at some later position. This can be only after the adjunction string (77), thus forming crossing adjunction sites. Contradiction. Now let us look at (72). Suppose that the left part begins with a proper binary string. Then $\wedge$ is preceded by some variable, and we can by the same argument not insert a closing bracket. Thus, we are left with only the following case.

(78) $\quad \langle\wedge\texttt{(p}\cdots\texttt{,}\cdots\texttt{)}\rangle$

Let us now see how we can derive an n-homogeneous string of order p, p large enough so that no trees of Type (c) can be used. The left periphery of this string consists in n opening brackets, which cannot be derived using rules of the form (78). Thus Type (b) with $\wedge$ in the left part (Case I) is ruled out. Type (b) (Case II) leads to (77) as the only possible adjunction string. But that cannot be used either, since there must be an opening bracket between the occurrence of $\wedge$ and $\texttt{p}$, since the formula is homogeneous. So, only trees of Type (a) can be used at the left periphery. They have this form.

(79) $\quad (*) \quad \langle \mathtt{(}\vec{x}\wedge, \mathtt{)}\rangle \qquad (\dagger) \quad \langle \mathtt{(}, \wedge\vec{x}\mathtt{)}\rangle$

Here $\vec{x}$ must be a formula. Make $n > pq$ where q is the number of nonterminals. Then it is the Type (†) that we need for the left periphery since $\vec{x}$ contains less than p symbols (and is not composed entirely of brackets). Now, if we have n opening brackets, there are q adjunction sites stacked for (†) inside each other. One pair of them has the same nonterminal, and from this we can get a (possibly derived) adjunction tree satisfying ① and ②.

Theorem 16 *There is no compositional TAG for* Bool$^\wedge$.

8 A Natural Language Example

The boolean language might be deemed to be irrelevant. However, consider the following transliteration into English:

(80)
$t(\mathtt{p})$ = `Jack sees a boy`
$t(\mathtt{(})$ = ε
$t(\mathtt{)})$ = ε
$t(\mathtt{0})$ = `who sees a girl`
$t(\mathtt{1})$ = `who sees a boy`
$t(\wedge)$ = `who sees no one and`
$t(\vee)$ = `who sees no one or`
$t(\neg)$ = `it is not the case that`

Now define

(81)
$s(\varepsilon)$:= `who sees no one.`
$s(a^\frown\vec{x}) := t(a)^\frown\square^\frown s(\vec{x})$

This gives us, for example,

(82)
$s(\mathtt{(p0\wedge(\neg p))})$ = `Jack sees a boy who sees a girl who sees no one and it is not the case that Jack sees a boy who sees no one.`

Consider the set $B = \{j\} \cup \{b\vec{x} : \vec{x} \in (\mathbb{0} \mid 1)^*\} \cup \{g\vec{x} : \vec{x} \in (\mathbb{0} \mid 1)^*\}$. Here j is Jack, $b\vec{x}$ is the boy number $\vec{x}$ and $g\vec{x}$ the girl number $\vec{x}$. Let $U \subseteq (\mathbb{0} \mid 1)^*$. Define $R(U)$ as follows.

(83) $$R(U) := \begin{cases} & \{\langle b\mathbb{0}\vec{x}, g\vec{x}\rangle : \vec{x} \in (\mathbb{0} \mid 1)^*\} \\ \cup & \{\langle g\mathbb{0}\vec{x}, g\vec{x}\rangle : \vec{x} \in (\mathbb{0} \mid 1)^*\} \\ \cup & \{\langle b1\vec{x}, b\vec{x}\rangle : \vec{x} \in (\mathbb{0} \mid 1)^*\} \\ \cup & \{\langle g1\vec{x}, b\vec{x}\rangle : \vec{x} \in (\mathbb{0} \mid 1)^*\} \\ \cup & \{\langle j, b\vec{x}\rangle : \vec{x} \in U\} \end{cases}$$

What can be shown is that the translation of $\mathtt{p}\vec{x}$ is true in $\langle B, j, R(U)\rangle$ (with $R(U)$ interpreting the relation of seeing and j interprets the constant 'Jack') iff $\vec{x} \in U$. Thus it turns out that the boolean language can be translated letter by letter into English preserving synonymy. Though the argument is not complete (for the reason that the English examples do away with brackets and so introduce ambiguity), it does serve to transfer Theorem 16 to English.

9 Discussion

A number of people including some reviewers have suggested that the proof is not conclusive for various reasons. One is that TAGs allow adjunction at different nodes. Thus we should make room for an infinite family of functions; whenever T is an adjunction tree and α a node address, there will be a unary function $f_{\alpha,T}$ adjoining T at position α. Second, as I noted above, I have explicitly excluded substitution from the list of operations. It could be added; however, since there are grammars based on substitution which use no adjunction (such as CFGs), it defeats the purpose to admit substitution into the list of admissible operations. Another shortcoming, pointed out above, is that we did not allow semantic operations to be partial. It would seem that relaxing our notion of grammar would help the matter. Yet, I have not been able to actually find a compositional TAG even when I allow partial semantic functions and infinite families. This is a strong indication that adjunction is not suited for the purpose.

Another problem that needs to be addressed is the identity of the meanings. Natural language semantics is fraught with difficulties, one of which is the problem of sense and reference. While that problem is a concern for the linguist, it is only of secondary relevance to us. This is because the semantics is assumed to be given at the outset, it is, if you will, empirical data. It is clear, though, that the semantics can be traded for another one modulo bijection. All we need to care about is whether or not two expressions are synonymous.[6] Though I am personally convinced that Montague's approach is more realistic (which assumes that every sentence starts out denoting a proposition and only denotes a truth value in a given possible world), nothing in particular hinges on that assumption. What I am really arguing here is rather that whatever languages we see around us, natural or formal, and whatever semantics we assume or diagnose them to have, adjunction is less suitable than bottom up creation of constituents, as for example in CFGs.

It is worth pointing out that there has been quite a series of proposals of computing semantic representations for TAGs, I mention here only [Gardent and Parmentier, 2005], [Gardent and Kallmeyer, 2003] and [Frank and van Genabith, 2001]. None of them have claimed their solution to be compositional, though. Recently, [Kallmeyer and Joshi, 2003] have claimed to have obtained a compositional semantics for TAGs. On a closer inspection the fundamental difference is that the semantics is computed *bottom up* from the derivation tree. However, standard derivations of TAGs are shown *top down*: the intermediate trees are center trees. Their yield is a string of the language in question. All other derivations proceed by (at least partially) adjoining to adjunction trees, and this includes the derivations proposed by [Kallmeyer and Joshi, 2003]. Thus, to the extent that the latter manages to establish a compositional semantics, it must proceed via intermediate trees of a different type.

There definitely are better ways to implement this idea. In fact, a more suitable form is that of a 2-LCFRS. In this type of grammar nodes represent not single strings (as in CFGs) but rather pairs of strings. This is because in tree terms, a

[6]As a reviewer points out, `2+2` and `2*2` are synonymous, but adding a `0` at the end of the first digit yields `20+2` in the first and `20*2` in the second example, which breaks the synonymy. To circumvent this s/he suggests to take expressions to be synonymous iff they can be substituted for each other in every context preserving observable meaning. To make this possible we need to have settled on a class of syntactic functions.

I find this suggestion puzzling. First, it requires a split between observable and what one might call substitutional meaning. Two expressions are declared substitutionally synonymous iff they can be substituted for each other in every context preserving observable meaning. Applied to the above it means that `2*2` and `2+2` are observably but not substitutionally synonymous.

What I object to is the confusion between form and meaning in this definition. Why should synonymy depend to surface syntax? What is the class of functions that we agree on beforehand? And, finally, what does the added notion of substitutional synonymy achieve for us? It is suggested that my notion of synonymy is based on substitution, but it is not. It is based on identity of meaning. No grammar enters here. Also, the fact that I use substitution at the level of analysis terms does not mean I am using adjunction at the surface syntax. One may of course consider adjunction at the analysis level, but I am not certain we shall gain more insight into the matter.

lower closed portion of an adjunction history actually defines an adjunction tree, which is a tree that has a single terminal x with category label identical to that of the root.[7] Given an adjunction tree T its yield is therefore of the form $\vec{x}A\vec{y}$, where A is a nonterminal and $\vec{x}$ and $\vec{y}$ are terminal strings. We now say that the pair that corresponds to this node is $\langle \vec{x}, \vec{y} \rangle$ and that its category is A.

The question is how we get that pair inductively by a bottom up procedure. Let u be a node in the derivation tree with label α, which is a basic adjunction tree. We assume that for each daughter v of u, the string pairs $\pi(v)$ are known. We need to find $\pi(u)$. The daughters are each adjoined to different nodes of α. For each node v, we define two strings, $\ell(v)$ and $r(v)$. If v is nonterminal, its category is the category it has in T. The strings are defined as follows. If v is terminal, then $\ell(v)$ is the string under v if v is to the left of the distinguished leaf, and ε otherwise. $r(v)$ is ε if v is to the left of the distinguished leaf and the string in T otherwise. If v is nonterminal, two cases arise. (1) No tree has been adjoined to v. Then $\ell(v) = r(v) = \varepsilon$. (2) Some tree has been adjoined. Let $\vec{x}A\vec{y}$ be its yield. Then $\ell(x) := \vec{x}$ and $r(y) := \vec{y}$. A must match the category of v. Now we define the string pair as follows. Let v_i, $i = 1, \ldots, m$, be an enumeration of the nodes such that (a) if $i < j$ then either v_i to the left of v_j or $v_i > v_j$. Then

$$\pi(u) = \langle \prod_{i=1}^{m} \ell(v_i), \prod_{i=m}^{1} r(v_i) \rangle \tag{84}$$

This is the string pair associated with the node u. The arity of the rule is maximally the number of nodes in the tree α. By collecting the rules corresponding to all possible local trees in a derivation tree we get the desired 2-LCFRS.

10 Conclusion

The present paper has established that there is no compositional semantics for the language of boolean expressions using only adjunction. Various remedies may be added, but it seems to me that everything points in the same direction: that adjunction is unsuited to carry the weight of semantic analysis. Moreover, in view of the fact that LCFRGs have the same weak generative strength as multicomponent TAGs and have been shown to be useful and easy to manage in formulating compositional semantic analyses (see among other [Calcagno, 1995]) it seems that adjunction should better be avoided when doing semantics.

[7] An exception is constituted by the root. Here the corresponding exponent is a actually best considered a string.

I conclude the paper with some open questions.

(Q1) Does the language Bool of interpreted boolean expressions have a factored string adjunction grammar?

(Q2) Does Bool have a string adjunction grammar with total semantic functions?

(Q3) Does Bool have a string adjunction grammar with partial semantic functions?

(Q4) Does Bool have a TAG with partial semantic functions using infinitely many rules?

For all of them I conjecture that the answer is negative.

References

[Calcagno, 1995] Mike Calcagno. A Sign–Based Extension to the Lambek Calculus for Discontinuous Constituents. *Bulletin of the IGPL*, 3:555 – 578, 1995.

[Frank and van Genabith, 2001] Anette Frank and Josef van Genabith. LL-based semantics for LTAG - and what it teaches us about LFG and LTAG. In Miriam Butt and Tracy Holloway King, editors, *Proceedings of the LFG'01 Conference*, University of Hong Kong, 2001. CSLI Online Publications.

[Gardent and Kallmeyer, 2003] Claire Gardent and Laura Kallmeyer. Semantic construction in Feature-Based TAG. In *Proceedings of the 10th Meeting of the European Chapter of the Association for Computational Linguistics, Budapest*, 2003.

[Gardent and Parmentier, 2005] Claire Gardent and Yannick Parmentier. Large scale semantic construction for tree adjoining grammar. In *Proceedings of Logical Aspects in Computational Linguistics*. Springer, 2005.

[Kallmeyer and Joshi, 2003] Laura Kallmeyer and Aravind Joshi. Factoring Predicate Argument and Scope Semantics: Underspecified Semantics with LTAG. *Research in Language and Computation*, 1:3 – 58, 2003.

[Kracht, 2003] Marcus Kracht. *The Mathematics of Language*. Number 63 in Studies in Generative Grammar. Mouton de Gruyter, Berlin, 2003.

[Kracht, 2006] Marcus Kracht. Partial Algebras, Meaning Categories and Algebraization. *Theoretical Computer Science*, 354:131–141, 2006.

[Kracht, 2007] Marcus Kracht. Lectures on interpreted languages and compositionality. Manuscript, UCLA, 2007.

[Martín-Vide and Păun, 1998] Carlos Martín-Vide and Gheorghe Păun. Structured Contextual Grammars. *Grammars*, 1:33–55, 1998.