ON SCOPE AND C-COMMAND

MARCUS KRACHT

1. Introduction

C-command and scope are notions used to talk about readings of a given sentence. In the sentence

(1) `Every man loves a woman.`

one says that in one reading `every` (or `every man`) "takes scope over" `a` (or `a woman`), and that in the other reading it is `a` that "takes scope over" `every`. Interchangeably, one says that in the first sentence `every man` (however, arguably not `every`) c-commands `a woman`, and that the situation is reversed in the second reading.

To have different readings is prima facie a matter of interpretation, or meaning, as I will consistently say. However, both scope and c-command happen to be defined in terms of structure. Scope is typically based on the string, while c-command is based on the tree structure that the sentence is thought to have. In what is to follow I shall be concerned with the exact ways in which tree based notions reflect string based ones, and vice versa; and with the question of how it is exactly that c-command and scope bear on interpretation. More precisely, I shall ask to what extent the c-command relation is sufficient in nailing down the structure inasfar as it matters for interpretive purposes. And thirdly I shall ask whether the different definitions given in the literature will make be different in this.

There is a connection between c-command and scope that has its roots both in the intellectual history of the notions themselves and the use that is made of them in the technical literature. Unfortunately, for neither notion is there an agreed definition. In this note I shall review some definitions and show that for all intents and purposes these definitions do give identical results.

2. Conventions

We assume that languages are formed over a basic set which we call *alphabet*. Elements of the alphabet are called *letters*. From letters we form *strings* by concatenation. The symbol of concatenation is $^\frown$. Actual letters or strings are written using typewriter font, like this: `Pxy→Qy`. Other symbols are just proxy for strings of a certain kind. Typically, I use $\vec{x}$, $\vec{u}$ to denote strings, but other wuthors have different habits. Concatenation is sometimes not even written. I write interchangeably

(2) `abbac`

and

(3) $\mathtt{a}^\frown\mathtt{b}^\frown\mathtt{b}^\frown\mathtt{a}^\frown\mathtt{c}$

Natural languages work a little differently. The letters of the language are actually the words, and the alphabet therefore should be called a dictionary. The fact that the words are themselves composed using letters of a different alphabet (unlike Chinese) may give rise to what is known as segmentation problems. These are problems to identify the basic words in a string. Suppose, by way of example, that we have a binary relation symbol `P` and a unary relation symbol `Pa` in addition to a constant `a` and a variable `x`. Then `Pax` is segmentable into a succession of `Pa` and `x` as well as a sequence of `P`, `a` and `x`. The presence of a blank eliminates most segmentation problems in written language. However, for the purpose of this note we assume that the language of the simple kind above: the basic constituents (alias words) are simply the letters of the alphabet.

3. Logic

The notion of scope is technically used only in connection with quantification. It is thus not even defined what the scope of a connective like $\rightarrow$ or $\vee$ is. The definitions of scope of a quantifier vary. [5] says the following. First, it is agreed that if $\mathcal{A}$ is a well-formed formulae and y a variable then $(y)\mathcal{A}$ is a well-formed formula as well.

In $((y)\mathcal{A}$, "$\mathcal{A}$" is the *scope* of the quantifier (y).

To understand this, one has to be reminded of the fact that in logic one first specifies an *alphabet* of symbols and then a set of so called *well-formed formulae* (= 'wffs'). Additional sets, such as the sets of *terms*, *variables* and so on are also defined. The definition typically amounts to a context-free grammar, where the sets are replaced by nonterminals standing in for these sets. Thus, if *Wff* is a nonterminal denoting the set of wfs, and *Var* a nonterminal for the set of variables, it is first agreed that we have a rule of the form

(4) $$\text{Wff} \rightarrow \text{`('}\ \text{Var}\ \text{')'}\ \text{Wff}$$

To say this is tantamount to saying that if $\vec{x}$ is in Var and $\vec{y}$ in Wff then the concatenation of '(' then $\vec{x}$ then ')' and then $\vec{y}$ is in Wff as well. It is customary to write this concatenation either as

(5) $$(^\frown\vec{x}^\frown)^\frown\vec{y}$$

thus using the $^\frown$ as a symbol of concatenation, or simply—suppressing this symbol—as

(6) $$(\vec{x})\vec{y}$$

Notice that there is no need for quotes anymore: we concatenate only strings!

Upon this one proceeds right away to the specification of meaning. Strings belonging to different sets will denote different things; in predicate logic, variables denote objects (under a given assignment), so do constants. Unary predicates denote sets of objects, binary predicates letters denote relations between objects. Finally, wffs denote truth values: these are *true* and *false*. We generally do not say, however, that an expression denotes *true* under a valuation but that it *is* true. What the definition (4) provides is a way to determine whether the expression formed using the syntactic rule is true or not. Thus, given we know the meaning of $\vec{x}$ and $\vec{y}$, we should be able to say what $(\vec{x})\vec{y}$ means. The definition (not shown here) is effectively the one standardly given. I paraphrase [5] as follows.

> $(y)\mathcal{A}$ is true under a given assignment σ if $\mathcal{A}$ is true under every assignment σ' which is different from σ at most in the value of y.

The notion of scope is not needed here; but we notice that Mendelsohn declares the scope of (y) to be $\mathcal{A}$. Scope is not even needed to talk about free and bound occurrences of a variable. Here we are told that an occurrence of a variable x is *bound* either if it is in an expression of the form (x) or in the scope of such an expression.

Machover in [4] gives a different account. He issues the following clause:

> (4) If $\mathbf{x}$ is a variable and b is an $\mathcal{L}$-*formula* then $\forall\mathbf{x}\boldsymbol{\beta}$ (the string obtained by concatenating a single occurrence of $\forall$, a single occurrence of $\mathbf{x}$ and the string $\boldsymbol{\beta}$, in this order) is an $\mathcal{L}$-formula.

Notice that this defines an altogether different language even if the same underlying letters were used! This is because $(\vec{x})$ is a different string than is $\forall\vec{x}$. Mendelsohn uses y to denote a string that is a variable, while Machover uses the symbol $\mathbf{x}$. We use $\vec{x}$ to say the same.

These differences are just stylistic; however, in Mendelsohn's language the universal quantifier is denoted by a pair of brackets enclosing the string, while in Machover's it is the symbol ∀ and no brackets are used.

(4) leaves nothing to desire in terms of explicitness. Greek letters are used as variables for strings that are $\mathfrak{L}$-formulae, while **x** is a variable over strings which are variables (of the language). Then we are told that

> A formula ∀**x**β constructed according to (4) is called a *universal* formula; here **x** is the *variable of quantification* and the string **x**β is the scope of the initial occurrence of the universal quantifier.

Here the scope includes every bound occurrence of the variable, which was not the case in the previous definition.

This definition fixes a few question of detail that the previous definition left implicit. Machover distinguishes between strings and occurrences of strings. We can make this distinction formal as follows. Let $\vec{x}$ be a string. An *occurrence* is a *pair* of strings. We say that $\langle \vec{v}, \vec{w} \rangle$ is *an occurrence of* $\vec{x}$ *in* $\vec{z}$ iff $\vec{z} = \vec{v}^\frown\vec{x}^\frown\vec{w}$. Thus, the notion of an occurrence makes sense only when we are talking about a given string. Say that string is `aabba`. In this string, `a` has three occurrences: $\langle \varepsilon, \texttt{abba} \rangle$ (where ε denotes the empty string), $\langle \texttt{a}, \texttt{bba} \rangle$ and $\langle \texttt{aabb}, \varepsilon \rangle$. `b` has only two occurrences: $\langle \texttt{aa}, \texttt{ba} \rangle$ and $\langle \texttt{aab}, \texttt{a} \rangle$. If you dislike the complication about pairs, think of an occurrence as a string with some portion underlined, such as this:

(7) $\texttt{qwe}\underline{\texttt{rty}}\texttt{uiop}$

This gives you both the host string (`qwertyuiop`) and the substring (`rty`). The occurrence is obtained by naming the part that is to the left of the line (`qwe`) and the part that is to the right (`uiop`).

To see that this care is needed let us look at the following formula:

(8) `∃xPx→∃x∀yxQy`

In this string, the quantifier `∃x` has two occurrences, so does the variable `x`: it occurs 4 times. We have to be exact as to which quantifier occurrence binds which variable occurrence. Talk of "the quantifier `∃x`" and "the variable `x`" is not enough in this circumstance. Thus, in the definition given by Mendelsohn, he ought to have said this:

> In $(y)\mathcal{A}$, "$\mathcal{A}$" is the *scope* of the *leftmost occurrence of* the quantifier (y).

As scopes are assumed to be disjoint from their quantifiers, this may be judged an innocent matter. For the only occurrence of (y) that is not in $\mathcal{A}$ is already the leftmost one. However, and this is mostly left implicit, the scope of (y) remains $\mathcal{A}$ even when we embed the formula into a larger one. Thus, in

(9) `(x)Px→¬(x)¬(y)xQy`

the formula `xQy` is the scope of the right most occurrence of `(x)`, but it is not the scope of the leftmost occurrence. The scope of the leftmost occurrence is larger. This is not innocent. For if we want to establish which variable is bound by which quantifier we must do the following: we need to be careful to talk only about occurrences and we need to be precise about scope. In detail, an occurrence C of a variable y is *bound* by an occurrence O of a quantifier (y) iff for every occurrence O' of the quantifier (y) whose scope contains C, its scope also contains O'. In linguistics, one reads "whose scope contains" simply as "which c-commands". And so we get the definition: *O binds C* iff (1) it c-commands C, (2) for every occurrence O' of (y): if $O' \neq O$, and O' c-commands C then O' c-commands O. Finally, replace reference to occurrences by reference to nodes, and label nodes by their strings, and you get a definition that says that a node labelled y is bound by the *minimal* c-commanding node labelled (y). This is a structure based equivalent of the string based notion.

The notion of scope, though defined in both books, actually plays no role, not even in the definition of bound variable. It is completely absent from other books that I have looked at ([2], [6]). This does not mean that it cannot be used; it means that it is largely irrelevant and that other concepts are used in its place.

4. Scope Generalised

Scope is basically a relation between occurrences of substrings in a string. To talk about scope means in effect to talk about the way the meaning of certain strings feeds the meaning of other strings. This is the link between these notions. In a nutshell, if a string $\vec{x}$ takes scope over some other string $\vec{y}$ then the meaning given to $\vec{y}$ feeds the meaning given to $\vec{x}$. We shall put this to work.

Above we have seen some examples. To see how this generalises, let us first see some more examples. The intention is that in

(10) `((A∨B)→C)`

(the occurrence of) `∨` is in the scope of (the occurrence of `→` but not conversely. How come? The idea is that the clauses that establish the meaning of an expression, here `((A∨B)→C)`, work inside out: the meaning of an (occurrence of a) substring is defined on the basis of the meaning of its parts. In the present example, the meaning of `(A∨B)` is defined on the basis of the meaning of `A` and the meaning of `B`. Next, the meaning of the entire string is defined on the basis of the meaning of `(A∨B)` and the meaning of `C`. That the meaning of `(A∨B)→C` indirectly depends also on the meaning of `A` is a consequence of the definition, and a welcome one. (Although the meaning of `A` can be a function of the meaning of `(¬A)` this does not mean that it is defined in this way. Sometimes outside-in-algorithms can be used, but definitions are always inside-out.) Thus, the algorithm for obtaining the meaning works inside-out: the meaning of a substring is determined on the basis of its immediate subexpressions. If we determine meanings according to this algorithm then if an occurrence of $\vec{x}$ is strictly contained in an occurrence of $\vec{y}$ then we shall compute the meaning of $\vec{x}$ before we compute that of $\vec{y}$.

Notice that some substrings are not assigned meaning; they are *non-constituents*. Such strings are `(A→`, for example. Constituents can be of different kind (term, variable, wff, and so on), in which case they may have different meanings. The meaning of a constituent is obtained simply by applying some recipe (= function) to the meaning of its immediate parts. There is a slight problem in that certain grammars allow an expression to be a part of itself in the sense that it can pass from being an expression of type A to an expression of type B. Such is the case with variables and terms. There is a rule Term → Var, and it says that any variable is also a term. In this case we require that the meaning of the string as a term is obtained in a direct and unambiguous manner from its meaning as a variable. This is just not really a problem, but it creates some uneasiness about the reduction of the tree structure to just strings.

Now let us look at a different string.

(11) `(A∨(B→C))`

In this string, the occurrence of `∨` now takes scope over `→`.

Now, for all intents and purposes, what matters for the determination of meaning is what the smallest constituent is that contains a given occurrence of a string $\vec{x}$. For it is this part of the string whose meaning is computed using the meaning of $\vec{x}$ directly. And so we assume the following definition:

> The *scope* of a constituent occurrence of a string $\vec{x}$ in a string $\vec{y}$ is the smallest constituent occurrence that properly contains the given occurrence of $\vec{x}$. (And it is $\vec{y}$ is $\vec{x} = \vec{y}$.)

However, we are most interested in the notion of "being in the scope of", which is rendered as follows.

> A constituent occurrence of a substring $\vec{z}$ of a given string $\vec{y}$ is in the scope of a constituent occurrence of a string $\vec{x}$ iff it is contained in the smallest constituent occurrence that contains the given occurrence of $\vec{x}$.

It is awkward to have to talk about occurrences of strings all the time. So we finally turn to trees. A string maps into a tree by taking as nodes all the constituent occurrences and draw lines for immediate containment. Then the previous definition can be rephrased for the corresponding constituent tree as follows.

> The *scope* of a non-root node x in a tree is the mother of x. y is *in the scope of* x if y is dominated by the mother of x, if x is not the root, by x otherwise.

Now, this way of doing this seems to introduce a circularity in the meaning algorithm since scope determines priority, so it ought not to contain loops or be reflexive. For that reason one often excludes all parts that x dominates, and the mother of y as well. This leads to a notion that is called idc-command:

> x idc-commands y iff x is not the root, x is incomparable with y, and the mother of x is above y.

Notice that this definition will make us go to the next node up rather than the next branching node up. I prefer this version because it is much clearer.

5. Scope Taking

One fundamental difference between formal languages and natural languages is that formal languages are *unambiguous.* They are also often *transparent.* To define these notions, let us recall that each of the definitions either explicitly or implicitly associates a (context free) grammar to the language. The grammar in turn can be said to *project* a structure over a string as follows. Define for an ordered tree T the *yield* of T, $Y(T)$ to be the concatenation of the letters at the leaves of T in the order specified by T. Each node x in the tree defines a constituent, the subtree headed by x, call it T_x, and this subtree therefore defines not only a substring $Y(T_x)$, but in addition an occurrence thereof. The *constituent structure projected over* $\vec{x}$ by T is precisely the set of all these occurrences, which are also called *constituents* of $\vec{x}$, or, somewhat more pedantically, *constituent occurrences of* $\vec{x}$ *under* T. So, let us take the string

(12) $$(\mathtt{A}\vee(\mathtt{B}\to\mathtt{C}))$$

Here, the strings that are (occurrences of) constituents are $\mathtt{A}$, $\mathtt{B}$, $\mathtt{C}$, $(\mathtt{B}\vee\mathtt{C})$ and the entire string. (I resist the temptation to present the occurrences of them ...) Now the string

(13) $$((\mathtt{A}\vee\mathtt{B})\to\mathtt{C})$$

gives us the constituents $\mathtt{A}$, $\mathtt{B}$, $\mathtt{C}$, $(\mathtt{A}\vee\mathtt{B})$ in addition to the entire string.

Now, it is customary to drop brackets between disjunctions. Thus, rather than write $((\mathtt{A}\vee\mathtt{B})\vee\mathtt{C})$ we write $\mathtt{A}\vee\mathtt{B}\vee\mathtt{C}$. If we do this it so happens that the same string can be the result of dropping brackets from two different strings. Or, if we do not consider them as the result of dropping brackets but rather as being genuinely generated by the grammar, then the situation is that we get two trees that the grammar generates which have the same yield. This means that the grammar is *ambiguous*. Formally we say that $\vec{x}$ *projects* T *in* G if T is a tree generated by G and $\vec{x} = Y(T)$. We say that G is ambiguous if there is a string that projects two different trees. This means that we cannot simply say that a certain (occurrence of a) substring is a constituent in the string. This may depend on the tree that we choose. So, in the present case we get the following alternative constituent structures:

(14) $$C_1 = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{A}\vee\mathtt{B}, \mathtt{A}\vee\mathtt{B}\vee\mathtt{C}\}$$

(15) $$C_2 = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{B}\vee\mathtt{C}, \mathtt{A}\vee\mathtt{B}\vee\mathtt{C}\}$$

In C_1, $\mathtt{A}\vee\mathtt{B}$ is a constituent, in C_2 it is not.

Let us make the assumption that there are no unary rules. Then the set of constituents uniquely characterises the tree. It does not, however, characterise uniquely the labels that we assign to the constituent strings. This must be stipulated as well. Finally, and thirdly, we assume that each letter is a constituent by itself. You have to take my word for it that neither of these requirements cause much concern; there are mild operations on the grammar that achieve this. Given that the meaning is assigned on the basis of the (labelled and ordered) tree, everything depends on the constituent structure, or the question which substring occurrences qualify as constituents. Once we reduced the matter to this question, we can finally the contribution that scope makes in this connection. By our definition, scopes are constituents that properly contain other constituents. Let us return to C_1 above. What are the scopes of this structure? They are $\mathtt{A}\vee\mathtt{B}$ and $\mathtt{A}\vee\mathtt{B}\vee\mathtt{C}$. The letters are excluded because scopes are constituents properly containing some constituent. Letters cannot contain anything, so they cannot be scopes. Since letters are constituents, there is no question here to answer. Therefore, by our stipulations, knowing all the scopes is tantamount to knowing the entire structure.

It is however not uncontroversial that the scope is a constituent. The next best definition is one that takes the scope of a constituent C to be the next constituent that it is contained it *excluding C itself.* Let this be the scope^1. Thus, in C_1, the scope of `→` consists in `(`, `A`, `B` and `)` and whatever constituents they make up. The disadvantage of this definition is that the remainder is not necessarily a string again. (Check that [4] did a very clever thing here!) On the other hand, if you know what the scope^1 of a constituent occurrence is you also know its scope and conversely. So the two are equivalent for the purpose of determining the sentence structure.

6. Safeness of Omitting Brackets

In formal languages, the conventions usually are that one inserts brackets around every constituent. Although this is tedious, it allows unique readability. It does even more: it guarantees that everything that looks like a constituent if taken by itself is a constituent in the string at hand. This is called *transparency.* If some substring has the same form as a constituent occurrence in some (!) tree projected by some (!) string, then it is a consituent in this string as well, no matter what the tree is that it projects. The grammar is thus unambiguous, since a given string can only project one tree. Let us call a bracketing *safe* if the resulting strings allow to recover the bracketing uniquely. Given a transparent grammar one then starts to drop brackets, as they get pretty tedious. We can, for example, drop the outermost brackets. Safeness is guaranteed: the full string is a constituent. Likewise, we do not enclose the letters into brackets. (Here is where segmentation problems may typically creep in!) From this point on, however, dropping brackets is not generally safe. Here is what can be done.

6.1. **Polish Notation.** Suppose the convention is that every connective is written before all of its arguments. So, we write `(→AB)` rather than `(A→B)`; a quantifier precedes its matrix, a generalised quantifier both its arguments, a predicate letter precedes all its argument terms, and so on. In this situation, all brackets can safely be dropped!

6.2. **Operator Precendence.** One agrees that certain operators bind stronger than others. It is generally agreed, for example, that `¬` binds stronger than `&`, which binds stronger than `→`, which in turn binds stronger than `∨`. The symbol that binds stronger will be outscoped by the weaker one. So, in the sequence `¬A∨B→C` the constituents are `¬A`, `A→B` and the full string. Or, in terms of bracketings, it derives from `((¬A)∨(B→C))`.

6.3. **Associativity.** It is agreed that a binary operator is either left or right associative. By that we mean the following. Suppose we have string

(16) $$\vec{x}_1 o \vec{x}_2 o \vec{x}_3 o \cdots o \vec{x}_n$$

where o is the operator, and the $\vec{x}_i$ are constituents. In case of left associativity the consituents are prefixes of each other, so we assume the following bracketing:

(17) $$(\cdots((\vec{x}_1 o \vec{x}_2) o \vec{x}_3) o \cdots o \vec{x}_n)$$

In case of right associativity suffixes

(18) $$(\vec{x}_1 o (\vec{x}_2 o (\vec{x}_3 o \ldots (\vec{x}_{n-1} o \vec{x}_n) \ldots)))$$

It does however also occur that we need not bother to fix any strategy at all. There is a big class of binary operations which are what is called *associative.* They satisfy that $(\vec{x}_1 o \vec{x}_2) o \vec{x}_3$ has the same meaning (!) as $\vec{x}_1 o (\vec{x}_2 o \vec{x}_3)$. For associative operation symbols different bracketings result in a different structure, but the meaning we associate with the different structures is the same. In this case we say that a given string is ambiguous but that the ambiguity is *spurious.* It vanishes upon mapping to the meaning.

7. C-Command

We are finally ready to see where this leads us for c-command. First, c-command is a relation on trees, and although one always refers to nodes using their labels ("the DP c-commands the QP") this is strictly speaking inappropriate. But it has the same status as talking about substrings as opposed to occurrences thereof. Once it is sufficiently clear what the distinction is we can ignore it. Now, there are different versions of c-command, and we shall discuss them in turn. Let R be a relation on T, where T is the set of nodes of the tree. Given $x \in T$, we put $xR := \{y : x\ R\ y\}$ and call this the *R-domain of x.*

The c-command domain of x is contained in a node $y \geq x$. This node we have taken to be either the mother of x (or the root if x is the root) or the least branching node above x (and the root if such a node does not exist). These versions are not different if every node which is not a leaf is branching. Under current assumptions in generative grammar this is the case. Also, we have assumed the rammar not to have unary rules, so again there is no distinction.

The next question is: are there any nodes to be excluded from the constituent headed by y? Most textbooks in linguistics agree that we want to exclude all nodes which are comparable to x. [1] and myself (see [3]) have argued that one should better not exclude any nodes from the constituent. Let us see whether this will make a difference. To assess this, we need to be clear about the question that we are raising. If I say that we are given the tree nodes, and we want to recover the dominance relation, then that is a pretty useless affair. The tree nodes are constituents of our string, so the nodes are already giving us the tree structure. To give you an example, let us take the string

(19) `Jack is a teacher and a hobby astronomer or a doctor.`

If I reveal the tree nodes, what could they possibly stand for if not strings? But as each string stands for a constituent, this is to reveal too much. So, what is in question is whether there exists a node in the tree corresponding to, say, `a hobby astronomer or a doctor`, or whether there is rather a node in the tree representing `a teacher and a hobby astronomer`. If we have the strings that correspond to tree nodes, the dominance order is simply inclusion, and the linear order is precedence. Thus, knowing the substrings is all there is to know anyway.

We assume therefore that we are given only the string. Additionally, we are given for each letter (or word, for that matter) the c-command domain in the form of the mother constituent. Once again, if we would give the c-command domain for each constituent this would be too much: then we would already have revealed the constituents (or practically all of them). To be a constituent other than the root is to stand in a c-command relation with someone else. So, we need something less trivial. Therefore, we say that we only reveal the c-command domain of each letter. This can be done in the form of just revealing the c-command relation *restricted to the set of letter occurrences.* This will be enough as soon as every constituent actually contains a letter as an immediate constituent. If not, additional information about other constituents must be made available. Now consider instead that you are given for each letter the c-command domain in the form of the remainder of the mother constituent minus the letter in question. It is a trivial matter to find the least constituent containing any given letter, and so we are back to the previous case. Thus, none of the two version are any different. This can be formalised thus.

Definition 1. *Let T be a set and $\geq$ a dominance relation on T. Then*

$$C(\geq) := \{\langle x, y\rangle : \textit{ for all } z > x\textit{: } z \leq z\} \tag{20}$$

$$C^{\circ}(\geq) := C(\geq) - \{\langle x, y\rangle : x \leq y \textit{ or } y \leq x\} \tag{21}$$

Proposition 2. *Let T be a set and $\geq$, $\geq'$ be dominance relations both with set $L \subsetneq T$ of leaves, such that every nonbranching node is a leaf (= in L). Assume that every constituent contains a leaf as immediate constituent. Then if $C(\geq) \cap L^2 = C(\geq') \cap L^2$, $\geq = \geq'$. Also, if $C^{\circ}(\geq) \cap L^2 = C^{\circ}(\geq') \cap L^2$, $\geq = \geq'$.*

Despite the rather complicated form in which this appears, this is actually quite useful for practical purposes. For the conditions are actually guaranteed by design of the grammar.

Observe that

$$C^{\circ}(\geq) \cap L^2 = (C(\geq) \cap L^2) - \{\langle x, x\rangle : x \in L\} \tag{22}$$

There is a final point to consider. The notion of c-command is actually symmetric. This is disturbing in many cases. For example, many people want it to be transitive but loop free. The latter comes down to being irreflexive. Thus if x c-commands y and y c-commands z we want that x also c-commands z; but we do not want that x c-commands x. In this case x c-commands y may actually be taken to mean something like: x needs y as input. Yet, under the above definitions, if x and y are sisters, they c-command each other. If c-command is transitive, x would c-command itself. This is unacceptable. So, we define our final notion, which is *asymmetric c-command*, *ac-command* for short. It is defined on the basis of c-command: x ac-commands y if (a) x c-commands y, (b) y does not c-command y.

Formally, this is done as follows. For a binary relation R, let $R^{\smile} := \{\langle y, x\rangle : x \; R \; y\}$. Further, put

$$A(R) := R - R^{\smile} \tag{23}$$

It can be verified that ac-command, based on any of the previous notions, is both transitive and irreflexive.

Now we ask the same question: suppose we have $A(C(\geq))$ restricted to the leaves—is it possible to reconstruct $\geq$? (It is easy to see that $A(C^{\circ}(\geq)) = A(C(\geq))$.) The answer is yes again. For this we proceed as follows. Let a be an occurrence of a letter with empty ac-domain. This is going to be of level 0. a is of level $n+1$ iff it does ac-command only letters of level n, and it does ac-command at least one such letter. Now, two letters of level $n+1$ are sisters iff they ac-command the same level n elements. Two letters of level 0 are sisters iff they are ac-commanded by the same letters of level 1.

References

[1] Chris Barker and Geoffrey Pullum. A theory of command relations. *Linguistics and Philosophy*, 13:1–34, 1990.

[2] Paul Gochet and Pascal Gribomont. *Logique*. Hermes, Paris, 1998.

[3] Marcus Kracht. Mathematical aspects of command relations. In *Proceedings of the EACL 93*, pages 241 – 250, 1993.

[4] Moshé Machover. *Set theory, logic and their limitations*. Cambridge University Press, Cambridge, 1996.

[5] Elliott Mendelsohn. *Introduction to Mathematical Logic*. Van Nostrand, New York, 1964.

[6] Wolfgang Rautenberg. *Einführung in die Mathematische Logik. Ein Lehrbuch mit Berücksichtigung der Logikprogrammierung*. Vieweg, Braunschweig/Wiesbaden, 1996.

Department of Linguistics, UCLA, 3125 Campbell Hall, Los Angeles, CA 90095-1543