# 10 Appendix 3 - Formalisation of Syntactic Structure

## 10.1 Formalisation

In all of the below there is a set of atomic symbols $A = \{a_1, a_2, \ldots, a_n\}$ and a sufficient supply of variable symbols, $V = \{v_1, v_2, \ldots\}$ that are distinct from each other and *A*. The brackets in the below are punctuation.

### 10.1.1 The syntax of trees

Let *L* be a set, then the set of *finite ordered finitely branching trees labelled from L* (or *ordered* trees for short), $T_L$, is built up recursively:

O1: if $l \in L$ then $(l, \varnothing) \in T_L$;

O2: if $l \in L$ and $b_1, \ldots, b_n \in T_L$ then $(l, (b_1, \ldots, b_n)) \in T_L$.

Given a set of variables, *V,* that are distinct from other symbols the set of *variable trees based on L and V,* $T_L^+$, is built up as follows:

V1: if $t \in L \cup V$ then $(t, \varnothing) \in T_L^+$;

V2: if $l \in L$ and $b_1, \ldots, b_n \in T_L^+ \cup V$ then $(l, (b_1, \ldots, b_n)) \in T_L^+$.

Given a variable tree, *t*, the variables, *v(t)*, can be re-collected as follows:

C1: if $t = (v, \varnothing)$ and $v \in V$ then $v(t) = \{v\}$;

C2: if $t = (l, (b_1, \ldots, b_n))$ then $v(t) = v((l, \varnothing)) \cup \bigcup_{i=1,\ldots,n}(v(b_i))$.

### 10.1.2 The syntax of rules

Given a set of syntaxes $SS = S_1, \ldots, S_n$, the *rules*, $R_{SS}$, are built up as follows:

R1: $(\Rightarrow) \in R_{SS}$;

R2: if $(a_1, \ldots, a_n \Rightarrow) \in R$ and $x \in T_A^+$ then $(a_1, \ldots, a_n, x \Rightarrow) \in R_{SS}$;

R3: if $(a_1, \ldots, a_n \Rightarrow) \in R$, $x \in T_A^+$ and $\{v_1, \ldots, v_n\} = v(x) \cup \bigcup_{i=1..m}(v(a_i))$ then $(a_1, \ldots, a_n \Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n) \in R_{SS}$.

### 10.1.3 The syntax of syntactic structures:

The set of *syntactic structures*, *S*, is built up as follows:

S1: $(\varnothing,\varnothing) \in S$;

S2: if $\{S_1, \ldots,S_n\} \subseteq S$ then $(\{S_1, \ldots,S_n\},\varnothing) \in S$;

S3: if $SS = \{S_1, \ldots,S_n\}$, $(SS,\varnothing) \in S$, and $\{r_1, \ldots, r_k\} \subseteq R_{SS}$ then $(SS, \{r_1, \ldots, r_k\}) \in S$.

If *(SS, R)* is a syntactic structure then *SS* is its *sub-syntaxes* and *R* is its *rules*. A syntax is *dependant* on a second if either this second syntactic structure is a sub-syntax of it or it is a sub-syntax of another syntactic structure that it is dependent on. Note that this definition includes *only* those syntactic structures that can be built up recursively like this, so there are no 'circular' structures with syntactic structures being dependent on themselves. If you draw a graph with syntactic structures as nodes and the relation of *being a sub-syntax of* being a directed arc then any collection of syntactic structures forms an acyclic digraph.

### 10.1.4 Generation from syntactic structures

Each syntactic structure, $S=(SS, R)$, generates a set of trees, $Gen(S)$, as follows:

G1: if $(\Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n) \in R$ and $t_1 \in Gen(S_1), \ldots, t_n \in Gen(S_n)$ then $(\Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n)[t_1, \ldots,t_n] \in Gen(S)$;

G2: if $(a_1, \ldots,a_m \Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n) \in R$, $t_1 \in Gen(S_1), \ldots, t_n \in Gen(S)$ such that $a_1(v_1/t_1)\ldots(v_n/t_n) \in Gen(S), \ldots, a_m(v_1/t_1)\ldots(v_n/t_n) \in Gen(S)$ and $s_1, \ldots,s_k$ are the $t_1, \ldots,t_n$ such that the corresponding $v_1, \ldots,v_n$ do not occur as subtrees of any of $a_1, \ldots,a_m$ then $(a_1, \ldots,a_m \Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n)[a_1(v_1/t_1)\ldots(v_n/t_n), \ldots, a_m(v_1/t_1)\ldots(v_n/t_n),s_1, \ldots,s_k] \in Gen(S)$;

where $A[b_1, \ldots,b_n]$ is the tree with $A$ as the top node label and $b_1, \ldots,b_n$ are the branches and $a(x/y)$ is the tree $a$ with every subtree $x$ replaced by subtree $y$.

Thus the trees *generated* by a syntactic structure like this are trees with rules as node labels and branches of the antecedents and other substitutions to be used in an application of the rule.

### 10.1.5 Production from trees

Each tree, $t \in Gen(S)$, *produces* a tree, $Prod(t)$, by recursively applying the rules at each node to the *production* of its branch trees and the substitutions from the generation of lower syntactic structures, thus:

(where in the below $t = (a_1, \ldots, a_m \Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n)[s_1, \ldots, s_m, t_1, \ldots, t_n]$)

P1: if $t = (\Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n)[t_1, \ldots, t_n]$ then $Prod(t) = x(v_1/t_1)\ldots(v_n/t_n)$;

P2: if $t = (a_1, \ldots, a_m \Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n)[s_1, \ldots, s_m, t_1, \ldots, t_k]$ and $s_1 = a_1(v_1/r_1)\ldots(v_n/r_n), \ldots, s_m = a_m(v_1/r_1)\ldots(v_n/r_n)$ where $\{t_1, \ldots, t_k\} \subseteq \{r_1, \ldots, r_n\}$ then $Prod(t) = x(v_1/r_1)\ldots(v_n/r_n)$.

### 10.1.6 Complete production

When a the process of *producing* from a tree, *t*, extends recursively downwards to its substitutions, we get a *complete production*, *ComProd(t)*, thus:

CP1: if $t = (\Rightarrow x)[]$ then $ComProd(t) = x$;

CP2: if $t = (\Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n)[t_1, \ldots, t_n]$ and $c_1 = ComProd(t_1), \ldots, c_n = ComProd(t_n)$ then $ComProd(t) = x(v_1/c_1)\ldots(v_n/c_n)$;

CP3: if $t = (a_1, \ldots, a_m \Rightarrow x, v_1 \in S_1, \ldots, v_n \in S_n)[s_1, \ldots, s_m, t_1, \ldots, t_k]$, $s_1 = a_1(v_1/r_1)\ldots(v_n/r_n), \ldots, s_m = a_m(v_1/r_1)\ldots(v_n/r_n)$ where $\{t_1, \ldots, t_k\} \subseteq \{r_1, \ldots, r_n\}$ and $c_1 = ComProd(r_1), \ldots, c_n = ComProd(r_n)$ then $ComProd(t) = x(v_1/c_1)\ldots(v_n/c_n)$.

A complete production of a tree has only symbols as the labels of its nodes and not rules.

### 10.1.7 Complete productive generation from syntactic structures

Each syntactic structure, *SS*, can recursively generate the productions of all trees in *Gen(SS)*, called the *complete productive generation*, *CPG(SS)*, thus:

CPG:if $t \in Gen(SS)$ then $ComProd(t) \in CPG(SS)$.

It is *CPG(SS)* that we usually associate with a syntax in normal logical parlance.

## 10.2 The Expressivity of Syntactic Structures

For any phrase structured grammar (PSG), there is a SS whose complete productive generation is the language generated by the PSG.

**Proof Outline**:

Let the PSG be defined in the usual manner by: a starting symbol, S; a set of variables, $V = \{S, A, B, C, \ldots\}$ ; a set of terminal symbols, $T = \{a, b, c, \ldots\}$ ; and a set

of production rules, P, of the form $g_i \rightarrow \overline{g_i}$ where $g_i, \overline{g_i} \in (V \cup T)^*$, where $L^*$ indicates the set of finite sequences made from the set $L$.

Basically I simulate the sequences of symbols with equivalent trees plus a count of how many non-terminals are left and then extract these to another syntactic level. Then the production rules of the PSG translate across in a straight forward manner.

Define a *SS*, $S_1 = (\varnothing, R_0)$, where $R_0$ includes the following rules:

PSG1: ⇒W(S,s(0)) - the starting symbol

PSG2: W(a,p(b,s(c)))⇒W(a,p(s(b),c)) - associativity of P

PSG3: W(a,p(b,0))⇒W(a,b) - answer of p

PSG4: W(a,m(s(b),s(c)))⇒W(a,m(b,c)) - associativity of m

PSG5: W(a,m(b,0))⇒W(a,b) - answer of m

PSG6: W(J(a,J(b,c)),x)⇒W(J(J(a,b),c),x) - associativity of J

PSG7: W(J(J(a,b),c),x)⇒W(J(a,J(b,c)),x) - associativity of J

The intention is that: W(a,n) represents a word, a, being processed in the PSG with n non-terminal symbols in it; S corresponds to the PSG's starting symbol; 0 is zero; s(n) is the successor function (i.e. n+1); p is for plus (i.e. p(a,b) is a+b); m is minus (i.e. m(a,b) represents a-b with a lower bound of zero); and J is the string concatenation operator (i.e. J(a,b) is the string of the a as a string followed by b as a string).

$R_0$ also includes rules to reflect the production rules in the PSG.

For each rule in the PSG of the form $S \rightarrow a_1 a_2 \ldots a_n$ there corresponds a SS rule of form

PSGi: $S \Rightarrow W(J(a_1, J(a_2, J(\ldots J(a_{n-1}, a_n)\ldots))), s(s(\ldots s(0)\ldots)))$,

where $s(s(\ldots s(0)\ldots))$ has the same depth as the number of non-terminals in $a_1 a_2 \ldots a_n$.

For each other rule in the PSG of the form $a_1 a_2 \ldots a_n \rightarrow c_1 c_2 \ldots c_p$ there corresponds a SS rule of the form

PSGj:

$$W(J((f, J(J(a_1, J(a_2, J(\ldots J(a_{n-1}, a_n)\ldots))), g)), s(s(\ldots s(0)\ldots))))$$
$$\Rightarrow W(J((f, J(J(c_1, J(c_2, J(\ldots J(c_{p-1}, c_p)\ldots))), g)), p(s(s(\ldots s(0)\ldots)), i)))$$

if it increases the number of non-terminal symbols in the PSG, where i is the increase in the number of non-terminals represented in the form $s(s(\ldots s(0)\ldots))$.

Finally for each other rule in the PSG of the form $a_1a_2\ldots a_n \rightarrow c_1c_2\ldots c_p$ there corresponds a SS rule of the form

PSGk:

$$W(J(\,(f, J(J(a_1, J(a_2, J(\ldots J(a_{n-1}, a_n)\ldots)))), g)), s(s(\ldots s(0)\ldots))))$$
$$\Rightarrow W(J(\,(f, J(J(c_1, J(c_2, J(\ldots J(c_{p-1}, c_p)\ldots)))), g)), m(s(s(\ldots s(0)\ldots)), i)))$$

if it decreases the number of non-terminal symbols in the PSG, where i is the decrease in the number of non-terminals represented in the form $s(s(\ldots s(0)\ldots))$.

If one rewrote the above SS rules by: w (n) for W(w,n); 1 for s(0); 2 for s(s(0)); …; n+1 for s(n); b+c for p(b,c); b-c for m(b,c); and ab for J(a,b) they would look like:

PSG1: $\Rightarrow$S (0)

PSG2: a (m+(n+1))$\Rightarrow$a ((m+1)+n)

PSG3: a (b+0)$\Rightarrow$a (n)

PSG4: a ((m+1)-(n+1))$\Rightarrow$a (m-n)

PSG5: a (n-0)$\Rightarrow$a (n)

PSG6: a(bc) (n)$\Rightarrow$(ab)c (n)

PSG7: (ab)c (n)$\Rightarrow$a(bc) (n)

PSGi: $S \Rightarrow a_1a_2\ldots a_n(n)$, where n is the number of non-terminals,

PSGj: $a_1a_2\ldots a_n(n) \rightarrow c_1c_2\ldots c_p(n+i)$

PSGk: $a_1a_2\ldots a_n(n) \rightarrow c_1c_2\ldots c_p(n-i)$

where n is the number of non-terminals on the LHS and i in the increase (respectively decrease) in the number of terminals due to the action of the PSG rule.

Then all the trees of the form a (0) correspond to the resulting words in the PSG (i.e. those with zero non-terminals in them).

Finally I define a second SS, $(S_1, R_0)$ with the single rule:

T: $W(a, 0) \Rightarrow a,\ (a \in S_0)$

to extract all the trees corresponding to the words generated by the PSG.

## 10.3 Flattening Syntactic Structures

For any finite collection of syntactic structures $\{S_0, \ldots, S_n\}$ which is closed w.r.t. sub-syntaxes, there is a corresponding *flattened* sequence of syntactic structures $(S_0', \ldots, S_k')$, such that:

1. for each $S_j$, there is a $S_k'$ with a generated set of trees that is identical;
2. for each i=1, …,k, free variables in rules in $S_i'$ only refer to one other $S_j'$, where i<j.

**Proof Outline**:

Reorder the collection $\{S_0, \ldots, S_n\}$ as $(T_0, \ldots, T_n)$ such that no $T_i$ has a sub-syntax with a greater index than itself, with the permutation $\theta: \{1,\ldots,n\} \rightarrow \{1,\ldots,n\}$. This is possible since $\{S_0, \ldots, S_n\}$ can be represented as an acyclic digraph.

Ensure that included in the atomic symbols $A = \{a_1, a_2, \ldots, a_n\}$, there is a distinct symbol for each of $(T_0, \ldots, T_n)$ plus one more, that are not used in any of the rules of any of $(T_0, \ldots, T_n)$, call these $t_0, \ldots, t_n, n$.

For each syntactic structure, $T_i = (SS_i, R_i)$, in $(T_0, \ldots, T_n)$, construct another $T_i' = (SS_i', R_i')$, to form a new sequence $(T_0', \ldots, T_n')$ such that:

(a) For each $T_j$ in $SS_i$, put $T_j'$ in $SS_i'$;

(b) for each rule $r_k = (a_1, a_2, \ldots, a_m \Rightarrow c, v_1 \in T_{n1}, \ldots, v_p \in T_{np})$ in $R_i$, put a rule $r_k' = (t_i(a_1'), t_i(a_2'), \ldots, t_i(a_m') \Rightarrow t_i(c'), v_1, \ldots, v_p \in T_{i-1})$ in $R_i$, where each $a_q'$ or c' is the result of replacing all occurrences of the $v_1, \ldots, v_p$ by $t_{n1}(v_1), \ldots, t_{np}(v_p)$. Also one extra rule, $(\Rightarrow v, v \in T_{i-1})$, to $R_i'$.

Essentially the generated results of the syntactic structures are accumulated up the sequence, with the origin of each preserved by the extra symbols.

Finally append a sequence of extra syntactic structures $(S_1', \ldots, S_n')$ which will correspond to the original collection, $\{S_0, \ldots, S_n\}$, such that $S_i' = (\{T_{\theta(i)}'\}, \{t_{\theta(i)}(v) \Rightarrow v\})$. This selects and strips the appropriate elements for the syntax.