Hierarchy in Fluid Construction Grammars

Joachim De Beule[1] and Luc Steels[1,2]
(1) Vrije Universiteit Brussel Artificial Intelligence Laboratory
(2) SONY Computer Science Laboratory - Paris.

March 20, 2005

Abstract

This paper reports further progress into a computational implementation of a new formalism for construction grammar, known as Fluid Construction Grammar (FCG). We focus in particular on how hierarchy can be implemented. The paper analyses the requirements for a proper treatment of hierarchy in grammar and then proposes a particular solution based on a new operator, known as the J-operator. The J-operator constructs a new unit as a side effect of the matching process.

1 Introduction

We have been researching a formalism for construction grammar and its implementation. The formalism is known as Fluid Construction Grammar (FCG), as it strives to capture the fluidity by which new grammatical constructions can enter or leave the language inventory. The implementation includes components for parsing a sentence into its semantic interpretation, and components that turn a semantic specification into a sentence via intermediary semantic and syntactic structures. Research is also going on how lexicons and grammars can be learned within the context of situated embodied language games. FCG uses many techniques from formal/computational linguistics, such as feature structures for representing syntactic and semantic information and unification as the basic mechanism for the selection and activation of rules. But the formalism and its processing components have a number of unique properties: All rules are bi-directional so that the same rules can be used for both parsing and production, and they can be flexibly applied so that ungrammatical sentences or meanings that are only partly covered by the language inventory, can be handled without catastrophic performance degradation.

So far FCG has dealt only with single layered structures without hierarchy. It is obvious however that natural languages make heavy use of hierarchy: a sentence can contain a noun phrase, which itself can contain another noun phrase, etc. This raises the question how hierarchy can be integrated in FCG without loosing the advantages of the formalism. This has turned out to be a very difficult problem, particularly if one wants to keep the bi-directionality of the rules, and it has taken almost a year to find its solution. The present paper discusses this solution. The reader is assumed to be familiar with the general concept of construction grammars (see [3], [2], [4] for introductions) and the basic ideas underlying Fluid Construction Grammars (see e.g. [6]).

In FCG, a construction always associates a semantic structure with a syntactic structure. The semantic structure contains various units (corresponding to lexical items or groupings of them) and semantic categorizations of these units. The syntactic structure contains also units (usually the same as the semantic structure) and syntactic categorizations. We first investigate the syntactic side of a construction before focusing on the semantic side and how the two are combined in a single construction.

2 Hierarchy in Syntax

2.1 Requirements

It is fairly easy to see what hierarchy means on the syntactic side and there is a long tradition of formalisms to handle it. To take an extremely simple example: the phrase "the big block" combines three lexical

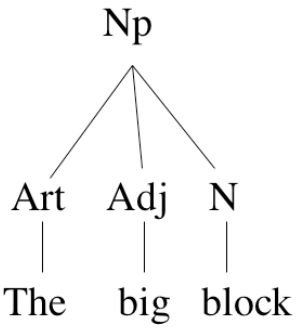

Figure 1: Hierarchical syntactic structure for a simple noun phrase

units to form a new whole. In terms of syntactic categories, we say that "the" is an article, "big" is an adjective, and "block" is a noun and that the combination is a noun phrase, which can function as a unit in a larger structure as in "the blue box next to the big block". Traditionally, this kind of phrase structure analysis is represented in graphs as in figure 1.

Using the terminology of construction grammar, we would say that "the big block" is a noun phrase construction of the type Art-Adj-Noun. "The blue box next to the big block" is another noun phrase construction of type NP-prep-NP. So constructions come in different types and subtypes. The units participating in a construction participate in particular grammatical relations (such as head, complement, determiner, etc.) but the relational view will not be developed here.

Various syntactic constraints are associated with the realization of a construction which constrain its applicability. For example, in the case of the Art-Adj-Noun construction, there is a particular word order imposed, so that the Article comes before the Adjective and the Adjective before the Noun. Agreement in number between the article and the noun is also required. In French, there would also be agreement between the adjective and the noun, and there would be different Art-Adj-Noun constructions with different word order patterns depending on the type of adjective and/or its semantic role ("une fille charmante" vs. "un bon copain").

When a parent-unit is constructed that has various subunits, there are usually properties of the subunits (most often the head) that are carried over to the parent-unit. For example, if a definite article occurs in an Art-Adj-Noun construction, then the noun phrase as a whole is also a definite noun phrase. Or if the head noun is singular than the noun phrase as a whole is singular as well.

It follows that the grammar must be able to express (1) what kind of units can be combined into a larger unit, and (2) what the properties are of this larger unit.

2.2 Syntactic Structure

The first step toward formalization of hierarchical constructions consists in determining what the syntactic structure looks like before and after the application of the construction.

Right before application of the construction in parsing, we have the following syntactic structure:

```
((top
 (syn-subunits (determiner modifier head))
 (form ((precedes determiner modifier)
        (precedes modifier head))))
 (determiner
    (syn-cat ((lex-cat article) (number singular)))
    (form ((stem determiner "the"))))
 (modifier
    (syn-cat ((lex-cat adjective)))
    (form ((stem modifier "big"))))
 (head
    (syn-cat ((lex-cat noun) (number singular)))
```

```
   (form ((stem head "block")))))                          (S1)
```

The lexicon contributes the different units and their stems. The precedes-constraints are directly detectable from the utterance.

The syntactic structure after the application of the Art-Adj-Noun construction is as follows:

```
((top
   (syn-subunits (np-unit)))
 (np-unit
   (syn-subunits (determiner modifier head))
   (form ((precedes determiner modifier)
          (precedes modifier head)))
   (syn-cat ((lex-cat NP) (number singular)))))
 (determiner
   (syn-cat ((lex-cat article) (number singular)))
   (form ((stem determiner "the"))))
 (modifier
   (syn-cat ((lex-cat adjective)))
   (form ((stem modifier "big"))))
 (head
   (syn-cat ((lex-cat noun) (number singular)))
   (form ((stem head "block")))))                          (S2)
```

This is a direct translation of the hierarchy shown in figure 1. As can be seen a new np-unit is inserted between the top-unit and the other units. The np-unit has the determiner, modifier and head-unit as subunits and also contains the precedence constraints involving these units. The lexical category of the new unit is NP and it inherits the number of the head-unit.

Let us now investigate what the constructions look like that license this kind of transformations.

2.3 The J-operator in syntactic parsing

In FCG a construction contains a semantic pole (left) and a syntactic pole (right). The application of a rule consists of a matching phase followed by a merging phase. For example, in parsing, the syntactic pole of a construction rule must match with the syntactic structure after which the semantic pole may be merged with the semantic structure to get a semantic structure. In production, the semantic pole must match with the semantic structure and the syntactic pole is then merged with the syntactic structure. We consider first the issue of parsing.

The key idea to handle hierarchy is to construct a new unit as a side effect of the matching and merging processes. Specifically, we can assume that, for the example under investigation, the syntactic pole should at least contain the following:

```
((?top
   (syn-subunits (== ?determiner ?modifier ?head))
   (form (== (precedes ?determiner ?modifier)
             (precedes ?modifier ?head))))
 (?determiner
   (syn-cat (== (lex-cat article) (number singular))))
 (?modifier
   (syn-cat (== (lex-cat adjective))))
 (?head
   (syn-cat (== (lex-cat noun) (number singular)))))
```

This matches with the syntactic structure given in (S1) and can therefore be used in parsing to test whether a noun-phrase occurs. But it does not yet create the noun-phrase unit.

We now propose that a new unit is created by the J-operator.[1] Units marked with the J-operator are ignored in matching. When matching is successful, the new unit is introduced and bound to the

[1] J denotes the first letter of Joachim De Beule who established the basis for this operator.

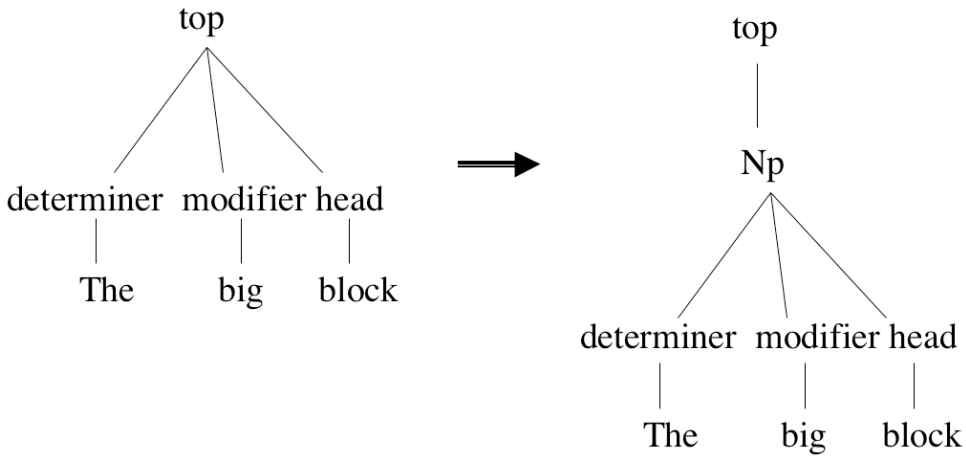

Figure 2: Restructuring performed by the J-operator

first argument of the J-operator. The second argument should be bound by the matching process and represents the parent unit from which the new unit should depend. The new unit can contain additional slot specifications, specified in the normal way, and all variable bindings resulting from the match are still valid. An example is shown below:

```
((?top
    (syn-subunits (== ?determiner ?modifier ?head))
    (form (== (precedes ?determiner ?modifier)
              (precedes ?modifier ?head))))
 (?determiner
    (syn-cat (==1 (lex-cat article) (number ?number))))
 (?modifier
    (syn-cat (==1 (lex-cat adjective))))
 (?head
    (syn-cat (==1 (lex-cat noun) (number ?number))))
 ((J ?new-unit ?top)
    (syn-cat (np (number ?number)))))            (S3)
```

Besides creating the new unit and adding its features, the overall structure should change also. Specifically, the new-unit is declared a subunit of the parent and all feature values specified by the unit with name equal to the second argument (i.e. ?top in the above) are moved to the new-unit. This way the precedence relations in the form-slot of the original parent or any other categories that transcend single units (like intonation) are automatically moved to the new unit as well. Thus the example syntactic structure (S1) for "the big block" above is transformed into the target structure (S2) by applying (S3).

Graphically the operation is illustrated in figure 2. Note that now the np-unit is itself a unit ready to be combined with others if necessary.

Note that now the np-unit is itself a unit ready to be combined with others if necessary.

2.4 Formal definition of the J-operator

The J-operator is a binary operator and is written as `(J ?new-unit ?parent)` (the variable names ?new and ?parent are arbitrary.)

Any unit-name in the left or right pole of a rule may be replaced by a J-operator. If so, the second argument (the `?parent` argument in the above example) should refer to another unit in the same rule.

Matching a source pattern containing a J-operator against a target structure is equivalent to matching the source pattern *without* the unit marked by the J-operator.

Merging a source pattern containing a J-operator with a target structure given a set of bindings containing at least a binding for the second variable `?parent` of the J-operator is defined as follows:

- Unless a binding is already present, the set of bindings is extended with a binding of the first argument `?new-unit` of the J-operator to a new constant unit-name `N`.
- A new source pattern is created by removing the unit marked by the J-operator from the original source pattern and adding a unit with name `N` whose slots contain the union of the slots of the unit marked by the J-operator and of the unit in the original source pattern with name equal to the second variable of the J-operator. This pattern is now merged with the target structure.
- The feature values of the unit in the original source pattern with name equal to the second variable of the J-operator are removed from the corresponding unit in the target structure and the new unit is made a subunit of this unit.

This is illustrated for the target structure (S1) and the source pattern (S3). The pattern (S3) matches with (S1) because the `(J ?new-unit ?top)` unit is ignored. The resulting bindings are:

```
((?top . top) (?determiner . determiner)
 (?modifier . modifier)  (?head . head)
 (?number . singular))                              (B1)
```

Before merging, these bindings are extended with a binding `(?new-unit . np-unit)`.

The new source pattern is given by

```
((?top
    (syn-subunits (== ?determiner ?modifier ?head))
    (form (== (precedes ?determiner ?modifier)
              (precedes ?modifier ?head))))
 (?determiner
    (syn-cat (==1 (lex-cat article) (number ?number))))
 (?modifier
    (syn-cat (==1 (lex-cat adjective))))
 (?head
    (syn-cat (==1 (lex-cat noun) (number ?number))))
 (np-unit
    (syn-cat (np (number ?number)))
    (syn-subunits (== ?determiner ?modifier ?head))
    (form (== (precedes ?determiner ?modifier)
              (precedes ?modifier ?head)))))
```

Merging this pattern with (S1) given the bindings (B1) results in

```
((top
    (syn-subunits (determiner modifier head))
    (form ((precedes determiner modifier)
           (precedes modifier head)))
 (np-unit
    (syn-subunits (determiner modifier head))
    (form ((precedes determiner modifier)
           (precedes modifier head)))
    (syn-cat ((lex-cat NP) (number singular)))))
 (determiner
    (syn-cat ((lex-cat article) (number singular)))
    (form ((stem determiner "the"))))
 (modifier
    (syn-cat ((lex-cat adjective)))
    (form ((stem modifier "big"))))
 (head
    (syn-cat ((lex-cat noun) (number singular)))
    (form ((stem head "block")))))
```

The last step changes the syn-subunits feature and delete the form feature of the top unit resulting in the structure (S2).

Note that the application of a rule now actually consists of three phases: first the matching of a pole against the corresponding structure, next the merging of the *same* structure with the altered pole as specified above and finally the normal merging phase (although here too J-operators might be involved.) However, for brevity we will sometimes neglect this and refer to the matching phase of a pole followed by the merging phase of the same but altered pole simply as matching phase and thus pretend as if this matching phase also introduces new units.[2]

2.5 J-operator in syntactic production

In syntactic production the syntactic structure after lexicon-lookup and before application of the Art-Adj-Noun construction could for example be given by:

```
((top
    (syn-subunits (determiner modifier head))
 (determiner
    (form ((stem determiner "the"))))
 (modifier
    (form ((stem modifier "big"))))
 (head
    (form ((stem head "block")))))              (S4)
```

As will be discussed shortly, in production the left (semantic) pole of the construction is matched with the semantic structure and the right pole (S3) has to be *merged* with the above syntactic structure. Here again the J-operator is used to create a new unit and link it from the top, yielding:

```
((top
   (syn-subunits (np-unit)))
 (np-unit
   (syn-subunits (determiner modifier head))
   (form ((precedes determiner modifier)
          (precedes modifier head)))
   (syn-cat ((lex-cat NP) (number singular)))))
 (determiner
   (syn-cat ((lex-cat article) (number singular)))
   (form ((stem determiner "the"))))
 (modifier
   (syn-cat ((lex-cat adjective)))
   (form ((stem modifier "big"))))
 (head
   (syn-cat ((lex-cat noun) (number singular)))
   (form ((stem head "ball")))))
```

[Usually the matching phase will result in a binding `np-unit` for thew `?new-unit` variable of the J-operator in (S3). Hence no new unit name has to be created while merging. But this is not absolutely necessary. The J-operator creates a new unit if necessary.

It is important to realise that merging may fail (and hence the application of the construction may fail), if there is an incompatibility between the slot-specification in the rule's pattern and the target structure against which the source pattern is matched. Indeed this is the function of ==1. It signals that there can only be one occurrence of the predicates that follow. If the slot-specification starts with == it means that the predicate can simply be added. The different forms of a slot-specification are summarised in the table below:

[2]It is even questionable whether matching should be done at all, since merging two structures that match is equivalent to matching them.

Notation	*Read as*	*Definition*
(a_1...a_n))	equal to	target must contain only given elements
(== a_1..a_n))	contains	given elements are members of target
(==1 a_1...a_n))	requires	elements with same predicate occur only once

In the example below the syn-cat of ?determiner requires `(==1 (lex-cat article) (number ?number))` This can be merged only if the lex-cat is none other than article and if the ?number is either not yet bound (in which case it becomes bound to the one contained in the structure) or if it is bound it is equal to the number value in the target structure.

```
((?top
    (syn-subunits (== ?determiner ?modifier ?head))
    (form (== (precedes ?determiner ?modifier)
              (precedes ?modifier ?head))))
 (?determiner
    (syn-cat (==1 (lex-cat article) (number ?number))))
 (?modifier
    (syn-cat (==1 (lex-cat adjective))))
 (?head
    (syn-cat (==1 (lex-cat noun) (number ?number))))
 ((J ?new-unit ?top)
    (syn-cat (np (number ?number)))))            (S3)
```

The same mechanisms also implement agreement as well as transfer from daughter to parent. ?number is the same for ?determiner, ?modifier and ?new-unit. If ?determiner and ?modifier have different number values, the merge is blocked. Assume for example that instead of the syntactic structure given in (S4) the following syntactic structure is provided, in which the head and determiner units are specified to have different number:

```
((top
 (syn-subunits (determiner modifier head))
 (determiner
    (syn-cat ((number singular)))
    (form ((stem determiner "the"))))
 (modifier
    (form ((stem modifier "big"))))
 (head
    (syn-cat ((number plural)))
    (form ((stem head "block")))))
```

The syntactic production mode requires that the right pole of the Art-Adj-Noun construction is merged with this structure. But because the pole specifies that both number values should be equal and because there can only be one number feature as specified by the `==1` operator, the merge will fail.

3 Hierarchy in Semantics

The mechanisms proposed so far implement the basic mechanism of syntactically grouping units together in more encompassing units. Any grammar formalism supports this kind of grouping. However constructions have both a syntactic and a semantic pole, and so now the question is how hierarchy is to be handled semantically.

A construction such as "the big block" not only simply groups together the meanings associated with its component units but also adds additional meaning, including what should be done with the meanings of the components to obtain the meaning of the utterance. FCG uses a Montague-style semantics (see [1] for an introduction), which means that individual words like "ball" or "big" introduce predicates and that constructions introduce second order semantic operators that say how these predicates are to be used (see [5] for a sketchy description of the system IRL that generates this kind of operators and how it is to be used in grammar).

For example, we can assume a semantic operator 'find-referent-1', which is a particular type of a find-referent operator that uses a quantifier, like [the], a property, like [big], and a prototype of an object, like [ball], to find an object in the current context. It would do this by filtering all objects that match the prototype, then filtering them further by taking the one who is biggest, and then taking out the unique element of the remaining singleton.

3.1 The J-operator in Semantic Parsing

The top level initial structure for "the big block" now looks as follows:

```
((top
   (meaning ((find-referent-1 obj det1 prop1 prototype1)
             (quantifier det1 [the])
             (property prop1 [big])
             (prototype prototype1 [ball])))))            (S5)
```

Lexicon-lookup introduces units for each of the parts of the meaning that can be covered[3]:

```
((top
    (sem-subunits (determiner modifier head))
    (meaning ((find-referent-1 obj det1 prop1 prototype1))))
 (determiner
    (referent det1)
    (meaning ((quantifier det1 [the]))))
 (modifier
    (referent prop1)
    (meaning ((property prop1 [big]))))
 (head
    (referent prototype1)
    (meaning ((prototype prototype1 [ball])))))          (S6)
```

The challenge now is to apply a construction which pulls out the additional part of the meaning that is not yet covered and constructs the appropriate new unit. Again the J-operator can be used with exactly the same behavior as seen earlier: The J-unit is ignored during matching, but after the match, a new unit is created, and its slots added. The new unit is linked to the parent and the parent slot specifications contained in the rule are moved to the new unit.

The semantic pole of the Art-Adj-Noun construction then looks as follows:

```
((?top
    (sem-subunits (== ?determiner ?modifier ?head))
    (meaning (== (find-referent-1 ?obj ?det1 ?prop1 ?prototype1))))
 (?determiner (referent ?det1))
 (?modifier (referent ?prop1))
 (?head (referent ?prototype1))
 ((J ?new-unit ?top) (referent ?obj)))
```

Application of this semantic pole to the semantic structure (S6) yields:

```
((top
   (sem-subunits (np-unit)))
 (np-unit
   (referent obj)
   (sem-subunits (determiner modifier head))
   (meaning ((find-referent-1 obj det1 prop1 prototype1))))
(determiner
```

[3]See section ? for an example of how the transformation between (S5) and (S6) can be accomplished using con-rules and the J-operator.

```
   (referent det1)
   (meaning ((quantifier det1 [the])))
(modifier
   (referent prop1)
   (meaning ((property prop1 [big]))))
(head
   (referent prototype1)
   (meaning ((prototype prototype1 [ball])))))
```

Various kinds of selection restrictions could be added in the form of semantic categories that may have been inferred using sem-rules. These selection restrictions would block the application of the construction while matching (in production) but also while merging (in parsing). For example, we could specialise the construction to be specific to the domain of physical objects (and hence physical properties and prototypes of physical objects) as follows:

```
((?top
    (sem-subunits (== ?determiner ?modifier ?head))
    (meaning (== (find-referent-1 ?obj ?det1 ?prop1 ?prototype1))))
 (?determiner (referent ?det1))
 (?modifier
    (referent ?prop1)
    (sem-cat (==1 (property-domain ?prop1 physical))))
 (?head
    (referent ?prototype1)
    (sem-cat (==1 (prototype-domain ?prototype1 physical))))
 ((J ?new-unit ?top)
    (referent ?obj)
    (sem-cat (==1 (object-domain ?obj physical)))))
```

giving the richer structure:

```
((top
   (sem-subunits (np-unit)))
 (np-unit
   (referent obj)
   (sem-subunits (determiner modifier head))
   (meaning ((find-referent-1 obj det1 prop1 prototype1)))
   (sem-cat ((object-domain obj physical))))
(determiner
   (referent det1)
   (meaning ((quantifier det1 [the]))))
(modifier
   (referent prop1)
   (meaning ((property prop1 [big])))
   (sem-cat ((property-domain prop1 physical))))
(head
   (referent prototype1)
   (meaning ((prototype prototype1 [ball])))
   (sem-cat ((prototype-domain prototype1 physical)))))
```

The Art-Adj-Noun construction as a whole therefore is implemented as follows:

```
(def-con-rule Art-Adj-Noun
    ((?top
        (sem-subunits (== ?determiner ?modifier ?head))
        (meaning (== (find-referent-1 ?obj ?det1 ?prop1 ?prototype1))))
     (?determiner (referent ?det1))
```

```
 (?modifier
    (referent ?prop1)
    (sem-cat (==1 (property-domain ?prop1 physical))))
 (?head
    (referent ?prototype1)
    (sem-cat (==1 (prototype-domain ?prototype1 physical))))
 ((J ?new-unit ?top)
    (referent ?obj)
    (sem-cat ((object-domain ?obj physical)))))
<-->
((?top
    (syn-subunits (== ?determiner ?modifier ?head))
    (form (== (precedes ?determiner ?modifier)
              (precedes ?modifier ?head))))
 (?determiner
    (syn-cat (==1 (lex-cat article) (number ?number))))
 (?modifier
    (syn-cat (==1 (lex-cat adjective))))
 (?head
    (syn-cat (==1 (lex-cat noun) (number ?number))))
 ((J ?new-unit ?top)
    (syn-cat (np (number ?number)))))
```

3.2 The J-operator in semantic parsing

The J-operator functions in semantic parsing exactly like in syntactic parsing. We can assume that the `?new-unit` variable of the J-operator has been bound to `np-unit` as a side effect of application of the syntactic pole. Thus, merging as defined in section 2.4 results in the extension of the semantic structure by a corresponding new unit. This unit is made a subunit of the top unit and the slot specifications of the top unit are moved to the new unit.

Suppose we start from the following semantic structure after lex-stem and sem-cat rules have been applied:

```
((top
   (sem-subunits (determiner modifier head)))
 (determiner (referent ?det1)
   (meaning ((quantifier ?det1 [the]))))
 (modifier
   (referent ?prop1)
   (meaning ((property ?prop1 [big])))
   (sem-cat ((property-domain ?prop1 physical-object))))
 (head (referent ?prototype1)
       (meaning ((prototype ?prototype1 [ball])))
       (sem-cat ((prototype-domain ?prototype1 physical-object)))))    (S7)
```

Merging now takes place to yield:

```
((top
    (sem-subunits (np-unit)))
 (np-unit
    (referent ?obj)
    (sem-subunits (determiner modifier head))
    (meaning ((find-referent-1 ?obj ?det1 ?prop1 ?prototype1)))
    (sem-cat ((object-domain ?obj physical))))
 (determiner
    (referent ?det1)
    (meaning ((quantifier ?det1 [the])))
```

```
(modifier
   (referent ?prop1)
   (meaning ((property ?prop1 [big])))
   (sem-cat ((property-domain ?prop1 physical-object))))
(head
   (referent ?prototype1)
   (meaning ((prototype ?prototype1 [ball])))
   (sem-cat ((prototype-domain ?prototype1 physical-object)))))
```

4 Implementing lex-stem rules

In the previous sections it was assumed that the application of lex-stem rules resulted in a structure containing separate units for every lexical entry. For example, it was assumed that the lex-stem rules transform the flat structure (S5) into (S6). However, to produce (S6) an additional transformation is needed that is normally not specified by the lex-stem rules itself. In this section we show how the the J-operator can be used to implement lex-stem rules that by itself introduce new units without the need for an additional transformation of the structure.

4.1 The J-operator in Lexical Production

In production the initial semantic structure is given by (S5):

```
((top
   (meaning ((find-referent-1 obj det1 prop1 prototype1)
             (quantifier det1 [the])
             (property prop1 [big])
             (prototype prototype1 [ball])))))            (S5)
```

Now consider the following lex-stem-rules

```
(def-con-rule [the]
    ((?top (meaning (== (quantifier ?x [the]))))
     ((J ?new-unit ?top)))
  <-->
  ((?top (syn-subunits (== ?new-unit)))
   (?new-unit (form (== (stem ?new-unit "the"))))))

(def-con-rule [big]
    ((?top (meaning (== (property ?x [big]))))
     ((J ?new-unit ?top)))
  <-->
  ((?top (syn-subunits (== ?new-unit)))
   (?new-unit (form (== (stem ?new-unit "big"))))))

(def-con-rule [block]
    ((?top (meaning (== (prototype ?x [block]))))
     ((J ?new-unit ?top)))
  <-->
  ((?top (syn-subunits (== ?new-unit)))
   (?new-unit (form (== (stem ?new-unit "block"))))))
```

As can be seen these rules only contain a J-operator in their left pole. This is because in production mode these rules have to create new units in both semantic and the syntactic domain. In interpretation mode however, they only have to create new units in the semantic domain. Indeed, in interpretation new units are created by the de-rendering and by the application of the morph-rules.

Matching the left pole of the [the]-rule with (S5) yields the bindings:

```
((?top . top) (?x . det1)).
```

As explained, merging this pole with (S5) extends these bindings with the binding `(?new-unit . [the]-unit)` and yields:

```
((top
   (sem-subunits ([the]-unit))
   (meaning ((find-referent-1 obj det1 prop1 prototype1)
             (property prop1 [big])
             (prototype prototype1 [ball]))))
 ([the]-unit
   (referent det1)
   (meaning ((quantifier det1 [the])))))
```

Note that, because of the working of the J-operator, a new unit is created which has absorbed the `(quantifier det1 [the])` part of the meaning from the top unit meaning. Applying the other two rules finally results in the structure (S6) as was desired.

The initial syntactic structure in production is empty and simply equal to `((top))`. But merging the right pole of the [the]-rule given the extended binding yields:

```
((top
   (syn-subunits ([the]-unit)))
 ([the]-unit
   (form ((stem [the]-unit "the")))))
```

Also applying the other rules finally gives:

```
((top
   (syn-subunits ([the]-unit [big]-unit [block]-unit)))
 ([the]-unit
   (form ((stem [the]-unit "the")))))
 ([big]-unit
   (form ((stem [the]-unit "big")))))
 ([block]-unit
   (form ((stem [the]-unit "block")))))
```

which is indeed equivalent to (S4).

4.2 The J-operator in Lexical Interpretation

In interpretation, the de-rendering and the application of morph-rules results in the syntactic structure (S4). However the semantic structure is now still empty and simply equal to `((top))`.

The syntactic structure (S4) is not changed when it is matched and merged with the right pole of the rules given above. However it does provide bindings for the unit-name variables in the left-pole of the rules. For example, when the right pole of the [the]-rule is matched with (S4) then `?top` is bound to `top` and `?new-unit` is bound to `[the]-unit`. Merging the left pole of the rule with the semantic structure given these bindings results in:

```
((top
   (sem-subunits ([the]-unit)))
 ([the]-unit
   (referent det1)
   (meaning ((quantifier ?x-1 [the])))))
```

Applying the other two rules finally yields

```
((top
   (sem-subunits ([the]-unit [big]-unit [block]-unit)))
 ([the]-unit (referent ?x-1)
```

```
    (meaning ((quantifier ?x-1 [the])))
  ([big]-unit
    (referent ?x-2)
    (meaning ((property ?x-2 [big]))))
  ([block]-unit
    (referent ?x-3)
    (meaning ((prototype ?x-3 [ball])))))
```

which is, apart from the sem-cat features which have to be added by sem-rules, indeed equivalent with (S7).

5 Conclusions

This paper introduced the J-operator and showed that it is a very elegant solution to handle hierarchical structure both in the syntactic and semantic domain. All the properties of FCG, and particularly the bi-directional character of rules, could be preserved. Further larger scale exercises are needed to further exercise the formal mechanisms proposed in the present paper.

6 Acknowledgement

The research in this paper was funded by the Sony Computer Science Laboratory and by grants from the ECAgents - EU FET program on Complex Systems. Joachim De Beule is funded as VUB teaching assistant.

References

[1] Dowty, D., R. Wall, and S. Peters (1981) Introduction to Montague Semantics. D. Reidel Pub. Cy., Dordrecht.

[2] Goldberg, A.E. 1995. *Constructions.: A Construction Grammar Approach to Argument Structure.* University of Chicago Press, Chicago

[3] Kay, P. and C.J. Fillmore (1999) Grammatical Constructions and Linguistic Generalizations: the What's X Doing Y? Construction. Language, may 1999.

[4] Michaelis, L. 2004. *Entity and Event Coercion in a Symbolic Theory of Syntax* In J.-O. Oestman and M. Fried, (eds.),Construction Grammar(s): Cognitive Grounding and Theoretical Extensions. Constructional Approaches to Language series, volume 2. Amsterdam: Benjamins.

[5] Steels, L. (2000) The Emergence of Grammar in Communicating Autonomous Robotic Agents. In: Horn, W., editor, Proceedings of European Conference on Artificial Intelligence 2000. Amsterdam, IOS Publishing. pp. 764-769.

[6] Steels, L. (2005) The Role of Construction Grammar in Fluid Language Grounding. AI Journal. Vol 164.

```
APPENDIX: Implementation Issues

;;; This is how J-matching and merging could be implemented (note that you do
;;; not need this code since it is integrated in the normal
match-structures and
;;; expand-structures.)

;; ;;; J-matching of a source to a pattern is defined as normal matching but with
;; ;;; the J-part removed from the pattern:
```

```
;; (defun J-match (source-struct pole &optional (bindings +no-bindings+))
;;   "Match source-struct with pole but after removing the J-unit part from pole."
;;   (let ((J (find-if #'(lambda (unit)
;;                       (and (consp (unit-name unit))
;;                            (eql (first (unit-name unit)) 'J)))
;;                   pole)))
;;     (match-structures (remove J pole)
;;                   source-struct
;;                   bindings)))

;; (defun maintain-hierarchy (J-unit Top-unit struct bindings)
;;   "Remove all feature-values from top-unit in struct and add them to the new
;; J-unit, and make J-unit a subunit of Top-unit (used in J-merge.)"
;;   (let ((source-top (structure-unit struct (lookup (unit-name Top-unit) bindings))))
;;     (dolist (feature (substitute-bindings bindings (unit-features Top-unit)))
;;       (dolist (value (feature-value feature))
;;      (setf (unit-feature-value source-top (feature-name feature))
;;            (delete value (unit-feature-value source-top (feature-name feature))
;;                    :test #'equalp))))
;;     (push (lookup (second (unit-name J-unit)) bindings)
;;        (unit-feature-value source-top (unit-name (get-subunits-feature Top-unit))))))

;; (defun J-merge (source-struct pole &optional bindings)
;;   "Merge source-struct with pole but with a new-unit for the J-unit in pole and
;; containing the specified features of the top-unit etc."
;;   (let* ((J (find-if #'(lambda (unit) (and (consp (unit-name unit))
;;                                            (eq (first (unit-name unit)) 'J)))
;;                   pole))
;;       (pattern-top (structure-unit pole (third (unit-name J)))))
;;     ;;; extend the bindings with a new constant for the new-unit variable in the
;;     ;;; J-unit (unless it is already in the bindings):
;;     (unless (lookup (second (unit-name J)) bindings)
;;       (setq bindings (extend-bindings (second (unit-name J))
;;                                       (new-const (second (unit-name J)))
;;                                       bindings)))
;;     ;;; Now do a normal expand structure but with the J-part in the pattern
;;     ;;; replaced by the new unit with features the union of the features of the
;;     ;;; J-part in the pole and the features of the top-unit specified in the
;;     ;;; pole:
;;     (let ((tasks (expand-structure
;;                `((,(second (unit-name J))
;;                   ,@(unit-features J)
;;                   ,@(unit-features pattern-top))
;;                  ,@(remove J (remove pattern-top pole)))
;;                source-struct
;;                bindings)))
;;       ;;; finally cleanup the result (e.g. make new-unit a a subunit of the top
;;       ;;; etc.):
;;       (dolist (task tasks tasks)
;;      (maintain-hierarchy J pattern-top (match-task-so-far task) (match-task-bindings task))))))

;; I. 'Simple' Example

;; poles of the con-rule:
(setq left-pole '((?top (subunits (== ?mp-u ?qp-u))
```

```
                    (meaning (== (IO-5 ?obj ?mp ?qp))))
                   (?mp-u (referent ?mp)
                    (sem-cat (== (phys-obj-prototype ?mp))))
                   (?qp-u (referent ?qp)
                    (sem-cat (== (quality-predicate ?qp))))
                   ((J ?new-unit ?top)
                    (referent ?obj)
                    (sem-cat (== (phys-obj ?obj))))))
(setq right-pole '((?top (subunits (== ?mp-u ?qp-u))
                     (form (== (precedes ?qp-u ?mp-u))))
                    (?mp-u
                     (syn-cat (==1 (lex-cat noun) (number ?number))))
                    (?qp-u
                     (syn-cat (==1 (lex-cat adjective) (number ?number))))
                    ((J ?new-unit ?top)
                     (syn-cat (== NP (number ?number))))))

;; initial semantic and syntactic structures:
(setq sem-struct '((top (subunits (mp-u qp-u))
                     (meaning ((IO-5 obj mp qp))))
                    (mp-u (referent mp)
                     (meaning ((prototype mp [ball])))
                     (sem-cat ((phys-obj-prototype mp))))
                    (qp-u (referent qp)
                     (meaning ((predicate qp [big])))
                     (sem-cat ((quality-predicate qp))))))
(setq syn-struct '((top (subunits (mp-u qp-u)))
                    (mp-u (form ((stem mp-u "ball"))))
                    (qp-u (form ((stem qp-u "big"))))))

;;; rule-application:
(setq bindings (first (match-structures  left-pole sem-struct)))

;; => (((?QP . QP) (?MP . MP) (?OBJ . OBJ) (?QP-U . QP-U) (?MP-U . MP-U) (?TOP . TOP)))
(setq task (first (expand-structure left-pole  sem-struct bindings)))
(setq sem-struct (match-task-so-far task))
(pprint-structure sem-struct)
;; ((TOP
;;     (SUBUNITS (NEW-UNIT-130)))
;;  (NEW-UNIT-130
;;     (MEANING ((IO-5 OBJ MP QP)))
;;     (REFERENT OBJ)
;;     (SEM-CAT ((PHYS-OBJ OBJ)))
;;     (SUBUNITS (MP-U QP-U)))
;;  (MP-U
;;     (MEANING ((PROTOTYPE MP [BALL])))
;;     (REFERENT MP)
;;     (SEM-CAT ((PHYS-OBJ-PROTOTYPE MP))))
;;  (QP-U
;;     (MEANING ((PREDICATE QP [BIG])))
;;     (REFERENT QP)
;;     (SEM-CAT ((QUALITY-PREDICATE QP)))))
(setq bindings (match-task-bindings task))
(setq task (first (expand-structure right-pole syn-struct bindings)))
(setq syn-struct (match-task-so-far task))
```

```
(pprint-structure syn-struct)
;; ((TOP
;;     (SUBUNITS (NEW-UNIT-130)))
;;  (NEW-UNIT-130
;;     (FORM ((PRECEDES QP-U MP-U)))
;;     (SUBUNITS (MP-U QP-U))
;;     (SYN-CAT (NP (NUMBER ?X-131))))
;;  (MP-U
;;     (FORM ((STEM MP-U ball)))
;;     (SYN-CAT ((LEX-CAT NOUN) (NUMBER ?X-131))))
;;  (QP-U
;;     (FORM ((STEM QP-U big)))
;;     (SYN-CAT (LEX-CAT ADJECTIVE) (NUMBER ?X-131)))))
(setq bindings (match-task-bindings task))

;; II. Example where the application of the rule is blocked by the merge

(setq sem-struct '((top (subunits (mp-u qp-u))
                        (meaning ((IO-5 obj mp qp))))
                   (mp-u (referent mp)
                    (meaning ((prototype mp [ball])))
                    (sem-cat ((phys-obj-prototype mp))))
                   (qp-u (referent qp)
                    (meaning ((predicate qp [big])))
                    (sem-cat ((quality-predicate qp))))))
(setq syn-struct '((top (subunits (mp-u qp-u)))
                   (mp-u (form ((stem mp-u "ball")))
                         (syn-cat (== (lex-cat Noun-2)))) ;;; <-
Assume the lex-cat of this unit is already known but incompatible
                   (qp-u (form ((stem qp-u "big"))))))

(setq bindings (first (match-structures left-pole sem-struct)))
;; => (((?QP . QP) (?MP . MP) (?OBJ . OBJ) (?QP-U . QP-U) (?MP-U . MP-U) (?TOP . TOP)))
(setq task (first (expand-structure left-pole sem-struct bindings)))
(setq sem-struct (match-task-so-far task))
(pprint-structure sem-struct)
;; ((TOP
;;     (SUBUNITS (NEW-UNIT-130)))
;;  (NEW-UNIT-130
;;     (MEANING ((IO-5 OBJ MP QP)))
;;     (REFERENT OBJ)
;;     (SEM-CAT ((PHYS-OBJ OBJ)))
;;     (SUBUNITS (MP-U QP-U)))
;;  (MP-U
;;     (MEANING ((PROTOTYPE MP [BALL])))
;;     (REFERENT MP)
;;     (SEM-CAT ((PHYS-OBJ-PROTOTYPE MP))))
;;  (QP-U
;;     (MEANING ((PREDICATE QP [BIG])))
;;     (REFERENT QP)
;;     (SEM-CAT ((QUALITY-PREDICATE QP)))))
(setq bindings (match-task-bindings task))
(setq task (first (expand-structure right-pole syn-struct bindings)))
;;; => NIL
```

```
;;; III. A more extended Example: "the big block" both in production and
;;; interpretation:

(clear-rule-set *con-rules*)
(def-con-rule Art-Adj-Noun-Cxn
     ((?top (sem-subunits (== ?determiner ?modifier ?head))
            (meaning (== (find-referent-1 ?obj ?det ?prop ?prototype))))
      (?determiner (referent ?det))
      (?modifier (referent ?prop)
                 (sem-cat (== (thing-property ?prop))))
      (?head (referent ?prototype)
             (sem-cat (== (physical-prototype ?prototype))))
      ((J ?new-unit ?top)
       (referent ?obj)
       (sem-cat (== (physical-object ?obj)))))
   <-->
   ((?top (syn-subunits (== ?determiner ?modifier ?head))
          (form (== (precedes ?determiner ?modifier)
                    (precedes ?modifier ?head))))
    (?determiner
     (syn-cat (==1 (lex-cat article) (number ?number))))
    (?modifier
     (syn-cat (==1 (lex-cat adjective))))
    (?head
     (syn-cat (==1 (lex-cat noun) (number ?number))))
    ((J ?new-unit ?top)
     (syn-cat (== np (number ?number))))))

;;; Production:
(setq *hypotheses* (list (cons '((top
                                  (sem-subunits (det-unit mod-unit head-unit))
                                  (meaning ((find-referent-1 obj-1
det-1 prop-1 prototype-1))))
                                 (det-unit
                                  (referent det-1)
                                  (meaning ((quantifier det-1 [the]))))
                                 (mod-unit
                                  (referent prop-1)
                                  (meaning ((property prop-1 [big])))
                                  (sem-cat ((thing-property prop-1))))
                                 (head-unit
                                  (referent prototype-1)
                                  (meaning ((prototype prototype-1 [block])))
                                  (sem-cat ((physical-prototype
prototype-1)))))
                               '((top
                                  (syn-subunits (det-unit mod-unit head-unit)))
                                 (det-unit
                                  (form ((stem det-unit "the"))))
                                 (mod-unit
                                  (form ((stem mod-unit "big"))))
                                 (head-unit
                                  (syn-cat ((number singular)))
                                  (form ((stem head-unit "block")))))))
```

```
(apply-rule-set *con-rules* '->)
(show-current)
;; Semantic structure:
;; ((X-199-229
;;     (MEANING ((FIND-REFERENT-1 OBJ-1 DET-1 PROP-1 PROTOTYPE-1)))
;;     (REFERENT OBJ-1)
;;     (SEM-CAT ((PHYSICAL-OBJECT OBJ-1)))
;;     (SEM-SUBUNITS (DET-UNIT MOD-UNIT HEAD-UNIT)))
;;  (DET-UNIT
;;     (MEANING ((QUANTIFIER DET-1 [THE])))
;;     (REFERENT DET-1))
;;  (HEAD-UNIT
;;     (MEANING ((PROTOTYPE PROTOTYPE-1 [BLOCK])))
;;     (REFERENT PROTOTYPE-1)
;;     (SEM-CAT ((PHYSICAL-PROTOTYPE PROTOTYPE-1))))
;;  (MOD-UNIT
;;     (MEANING ((PROPERTY PROP-1 [BIG])))
;;     (REFERENT PROP-1)
;;     (SEM-CAT ((THING-PROPERTY PROP-1))))
;;  (TOP
;;     (SEM-SUBUNITS (X-199-229))))
;; Syntactic structure:
;; ((X-199-229
;;     (FORM ((PRECEDES DET-UNIT MOD-UNIT) (PRECEDES MOD-UNIT HEAD-UNIT)))
;;     (SYN-CAT (NP (NUMBER ?X-200)))
;;     (SYN-SUBUNITS (DET-UNIT MOD-UNIT HEAD-UNIT)))
;;  (DET-UNIT
;;     (FORM ((STEM DET-UNIT the)))
;;     (SYN-CAT ((LEX-CAT ARTICLE) (NUMBER SINGULAR))))
;;  (HEAD-UNIT
;;     (FORM ((STEM HEAD-UNIT block)))
;;     (SYN-CAT ((NUMBER SINGULAR) (LEX-CAT NOUN))))
;;  (MOD-UNIT
;;     (FORM ((STEM MOD-UNIT big)))
;;     (SYN-CAT ((LEX-CAT ADJECTIVE))))
;;  (TOP
;;     (SYN-SUBUNITS (X-199-229))))

;;; II. Interpretation
(setq *hypotheses* (list (cons '((top
                                  (sem-subunits (det-unit mod-unit head-unit)))
                                 (det-unit
                                  (referent det-1)
                                  (meaning ((quantifier det-1 [the]))))
                                 (mod-unit
                                  (referent prop-1)
                                  (meaning ((property prop-1 [big])))
                                  (sem-cat ((thing-property prop-1))))
                                 (head-unit
                                  (referent prototype-1)
                                  (meaning ((prototype prototype-1 [block])))
                                  (sem-cat ((physical-prototype prototype-1)))))
                               '((top
                                  (syn-subunits (det-unit mod-unit head-unit))
                                  (form ((precedes det-unit mod-unit) (precedes mod-unit head-uni
```

```
                                    (det-unit
                                     (syn-cat ((lex-cat article) (number singular)))
                                     (form ((stem det-unit "the"))))
                                    (mod-unit
                                     (syn-cat ((lex-cat adjective)))
                                     (form ((stem mod-unit "big"))))
                                    (head-unit
                                     (syn-cat ((lex-cat noun) (number singular)))
                                     (form ((stem head-unit "block"))))))))
(apply-rule-set *con-rules* '<-)
(show-current)
;; ((X-199-231
;;     (MEANING ((FIND-REFERENT-1 ?X-195 ?X-196 ?X-197 ?X-198)))
;;     (REFERENT ?X-195)
;;     (SEM-CAT ((PHYSICAL-OBJECT ?X-195)))
;;     (SEM-SUBUNITS (DET-UNIT MOD-UNIT HEAD-UNIT)))
;;  (DET-UNIT
;;     (MEANING ((QUANTIFIER DET-1 [THE])))
;;     (REFERENT DET-1))
;;  (HEAD-UNIT
;;     (MEANING ((PROTOTYPE PROTOTYPE-1 [BLOCK])))
;;     (REFERENT PROTOTYPE-1)
;;     (SEM-CAT ((PHYSICAL-PROTOTYPE PROTOTYPE-1))))
;;  (MOD-UNIT
;;     (MEANING ((PROPERTY PROP-1 [BIG])))
;;     (REFERENT PROP-1)
;;     (SEM-CAT ((THING-PROPERTY PROP-1))))
;;  (TOP
;;     (MEANING ((FIND-REFERENT-1 ?X-232 DET-1 PROP-1 PROTOTYPE-1)))
;;     (SEM-SUBUNITS (X-199-231))))
;; Syntactic structure:
;; ((X-199-231
;;     (FORM ((PRECEDES DET-UNIT MOD-UNIT) (PRECEDES MOD-UNIT HEAD-UNIT)))
;;     (SYN-CAT (NP (NUMBER SINGULAR)))
;;     (SYN-SUBUNITS (DET-UNIT MOD-UNIT HEAD-UNIT)))
;;  (DET-UNIT
;;     (FORM ((STEM DET-UNIT the)))
;;     (SYN-CAT ((LEX-CAT ARTICLE) (NUMBER SINGULAR))))
;;  (HEAD-UNIT
;;     (FORM ((STEM HEAD-UNIT block)))
;;     (SYN-CAT ((LEX-CAT NOUN) (NUMBER SINGULAR))))
;;  (MOD-UNIT
;;     (FORM ((STEM MOD-UNIT big)))
;;     (SYN-CAT ((LEX-CAT ADJECTIVE))))
;;  (TOP
;;     (SYN-SUBUNITS (X-199-231))))


;;; IV. We can also get rid of external manipulation after the application of
;;; the lex-stem-rules?
(clear-rule-set *con-rules*)
(def-con-rule the
     ((?top (meaning (== (quantifier ?x [the]))))
      ((J ?new-unit ?top)))
   <-->
```

```
   ((?top (syn-subunits (== ?new-unit)))
    (?new-unit (form (== (stem ?new-unit "the"))))))
(def-con-rule big
     ((?top (meaning (== (property ?x [big]))))
      ((J ?new-unit ?top)))
   <-->
   ((?top (syn-subunits (== ?new-unit)))
    (?new-unit (form (== (stem ?new-unit "big"))))))
(def-con-rule block
     ((?top (meaning (== (prototype ?x [block]))))
      ((J ?new-unit ?top)))
   <-->
   ((?top (syn-subunits (== ?new-unit)))
    (?new-unit (form (== (stem ?new-unit "block"))))))


;;; Production:
(set-sem ((top (meaning ((find-referent-1 obj-1 det-1 prop-1 prototype-1)
                         (quantifier det-1 [the])
                         (property prop-1 [big])
                         (prototype prototype-1 [block])))))))


(apply-rule-set *con-rules* '->)
(show-current)

;; Semantic structure:
;; ((TOP
;;     (MEANING ((FIND-REFERENT-1 OBJ-1 DET-1 PROP-1 PROTOTYPE-1)))
;;     (SEM-SUBUNITS (X-338-356 X-335-355 X-332-354)))
;;  (X-332-354
;;     (MEANING ((QUANTIFIER DET-1 [THE]))))
;;  (X-335-355
;;     (MEANING ((PROPERTY PROP-1 [BIG]))))
;;  (X-338-356
;;     (MEANING ((PROTOTYPE PROTOTYPE-1 [BLOCK])))))
;; Syntactic structure:
;; ((TOP
;;     (SYN-SUBUNITS (X-338-356 X-335-355 X-332-354)))
;;  (X-332-354
;;     (FORM ((STEM X-332-354 the))))
;;  (X-335-355
;;     (FORM ((STEM X-335-355 big))))
;;  (X-338-356
;;     (FORM ((STEM X-338-356 block)))))

;;; Interpretation: Now the subunits are already present in the syntactic
;;; structure because of parsing and the application of morph-rules:
(setf *hypotheses* (list (cons '((top)) ;; but not in the semantic structure
                               '((top (syn-subunits (the-unit big-unit block-unit)))
                                 (the-unit (form ((stem the-unit "the"))))
                                 (big-unit (form ((stem big-unit "big"))))
                                 (block-unit (form ((stem block-unit "block")))))))))

(apply-rule-set *con-rules* '<-)
```

```
(show-current)

;; Semantic structure:
;; ((TOP
;;     (SEM-SUBUNITS (BLOCK-UNIT BIG-UNIT THE-UNIT)))
;;  (BIG-UNIT
;;     (MEANING ((PROPERTY ?X-363 [BIG]))))
;;  (BLOCK-UNIT
;;     (MEANING ((PROTOTYPE ?X-364 [BLOCK]))))
;;  (THE-UNIT
;;     (MEANING ((QUANTIFIER ?X-362 [THE])))))
;; Syntactic structure:
;; ((TOP
;;     (SYN-SUBUNITS (THE-UNIT BIG-UNIT BLOCK-UNIT)))
;;  (BIG-UNIT
;;     (FORM ((STEM BIG-UNIT big))))
;;  (BLOCK-UNIT
;;     (FORM ((STEM BLOCK-UNIT block))))
;;  (THE-UNIT
;;     (FORM ((STEM THE-UNIT the)))))
```