A First Encounter with Fluid Construction Grammar

Luc Steels

This paper is the authors' draft and has now been officially published as:
Luc Steels (2011). A First Encounter with Fluid Construction Grammar. In Luc Steels (Ed.), *Design Patterns in Fluid Construction Grammar*, 31–68. Amsterdam: John Benjamins.

Abstract

This chapter introduces the main mechanisms available in FCG for representing constructions and transient structures. It sketches the process whereby constructions are applied to expand transient structures and illustrates how templates are used to define constructions in a more abstract and modular way. Lexical constructions are used as the main source of examples.

1. Introduction

Fluid Construction Grammar (FCG) is a formalism for defining the inventory of lexical and grammatical conventions that a language user needs to know and the opserations with which this inventory is used to parse and produce sentences. FCG supports two ways to define constructions. One can use templates which abstract away from many details to highlight the linguistic content of a construction. Templates allow designers to implement a particular design pattern found in human languages, such as phrase structure, field topology, agreement systems, unmarked forms, argument structure, etc. Templates specify aspects of a construction which are then assembled into an 'operational' construction, i.e. a construction that contains all the details necessary for driving parsing and production.

This chapter provides a first survey of the main elements available for representing linguistic structures and for orchestrating constructional processing at the operational level. Although linguists will mostly use templates for defining grammar fragments of specific languages, it is nevertheless useful to know what the building

blocks of FCG are at the operational level. This is of course also necessary if one wants to implement new templates.

The first section introduces the tools available for representing transient structures. FCG uses feature structures (Carpenter, 2002) which are widely used by many linguistic formalisms. A transient structure consists of a set of units, features associated with these units, and values for these features. FCG splits the features into a semantic and a syntactic pole to improve readability and make processing more efficient. Feature structures are not only used to represent transient structures but also constructions.

Constructions are examined in the second section. FCG uses techniques from unification-based grammars to implement constructional processing (Kay, 1986). The application of a construction proceeds by matching the conditional pole (the semantic pole in production or the syntactic pole in parsing) against the current transient structure and by adding the information contained in the contributing pole (the syntactic pole in production or the semantic pole in parsing). This section looks at how variables and additional operators are available to make constructions more abstract, and at the J-operator which is the main primitive for building hierarchical structure.

The chapter ends with a discussion of templates, which is the primary way through which the definition of constructions is made more modular and hence easier to read and implement. Some templates create the skeleton of a construction whereas others add more components in order to handle different issues, such as the linking of constituent meanings, agreement relations, etc.

2. Representing Transient Structures

A transient structure contains all the information that is needed either to parse or produce a sentence. Parsing is not just the construction of a syntactic structure but the full reconstruction of the meaning of a sentence, including the proper linking of the meanings contributed by the different lexical items and constructions used. Production is not just the generation of a random syntactic structure but the transduction of meaning into a fully specified surface form. Human languages are known to employ a lot of constraints from many different levels of linguistic analysis (pragmatic, semantic, syntactic, morphological, phonological and phonetic) and so transient structures encompass all these levels. They typically contain dozens of units and hundreds of features.

2.1. Units

Transient structures are decomposed into a set of *units*, which correspond to individual words, morphemes, or constituents. Each unit has a set of *features* that hold information about the unit. A unit has both a semantic and a syntactic pole.

- The *semantic pole* contains features that concern the meaning and communicative function aspects of a unit, including what arguments are available to combine this meaning with meanings supplied by other units. The semantic pole also contains pragmatic and semantic categorizations and a list of the semantic subunits of the unit.

- The *syntactic pole* contains features concerning the syntactic side of a unit. This includes constraints on the form of the utterance (for example how the forms of the subunits are to be ordered sequentially), as well as syntactic, morphological and phonological categorizations of the unit and its possible syntactic subunits.

Both poles may also have *footprints* recording which constructions participated in creating them, as explained later.

The semantic and syntactic poles are typically displayed separately, in the sense that the semantic poles for all units are grouped together on the left side and the syntactic poles for all units are grouped together on the right side. (See Figure 3 and 2.) Such representations are sometimes called *coupled feature structures*. It is not the only representation possible, because all features of a unit could also be displayed together in one big list, but this bipolar arrangement is more convenient for examining complex transient structures.

Each unit in a transient structure has a unique *name* which can be used to refer to the unit. Unit names are also very useful when inspecting a feature structure and are the first example of FCG-symbols that are used liberally in constructions and constructional processing.

An *FCG-symbol* consists of a set of characters without spaces. There is no restriction on the characters, and to avoid confusion and typing errors, no distinction is made for letter case. `nominal-phrase` is equivalent to `Nominal-phrase` or `Nominal-Phrase`. As soon as a symbol is used, it is known to the FCG-interpreter. For example, in order to specify the syntactic subunits of a unit we can simply list the names of these subunits. One special class of FCG-symbols acts as variables, which are conventionally denoted by putting a question mark in front, as in

?phrase or ?gender. Variables play a crucial role in constructional processing as is explained in the next section.

Figure 3 contains the graphical representation of a simplified transient structure. The relevant constructions for this example are described in more detail in a later chapter of this book (Steels, 2011). Such graphical representations are used for inspecting and browsing through linguistic processes which quickly become very complicated (see Bleys et al., 2011). By clicking on units, detail becomes visible or is hidden. Even a simple phrase may contain large amounts of information, both on the syntactic and semantic side, therefore transient structures can take up several pages when displayed in full. Figure 3 shows a small, simplified example. An enlarged display of the same structure is shown in Figure 2. The underlying tree of units and subunits is shown in figure 1. Much more information, for example the functional structure, is contained in the various features attached to each node of this tree.

The feature structure shown in Figure 3 decomposes the phrase, "the mouse", into three units: nominal-phrase-12 for the nominal phrase as a whole, the-11 for the unit that introduces the word "the" and mouse-12 for the unit that introduces the word "mouse". The article and the noun hang from a unit, called nominal-phrase-12, which itself hangs from a unit called top. The top unit acts like a buffer, containing input materials that have not been treated yet. (There are none here.) The indices occurring after names of units (12 in nominal-phrase-12 for example) have been generated by the FCG-interpreter.

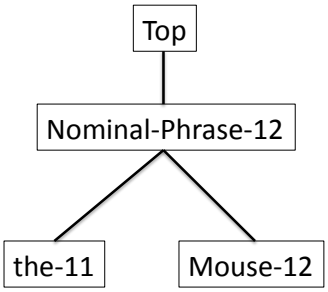

Figure 1. *The constituent structure tree underlying the transient structure shown in 2.*

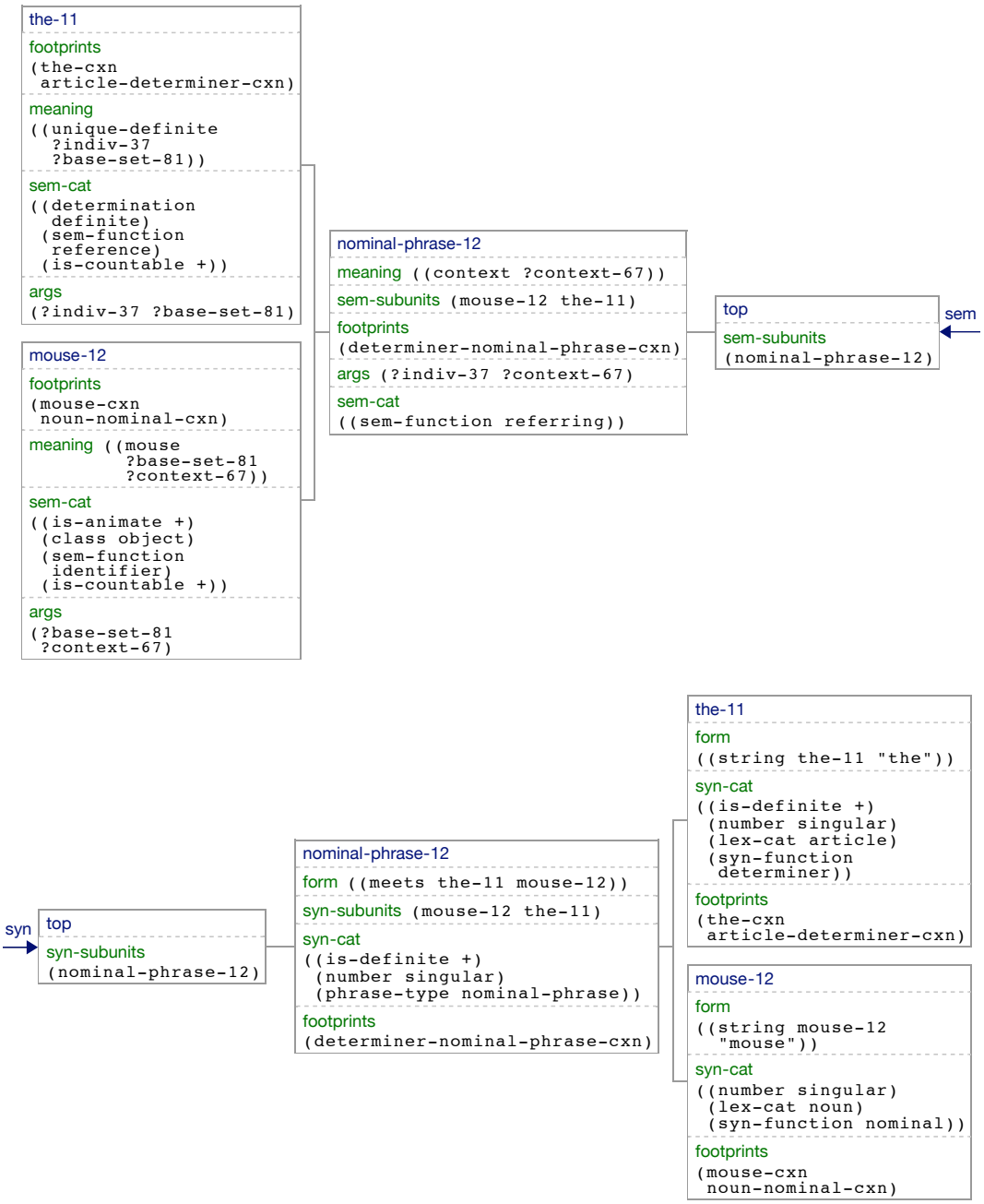

Figure 2. *Zooming in on the semantic (top) and syntactic (bottom) poles of the transient structure shown in figure 3.*

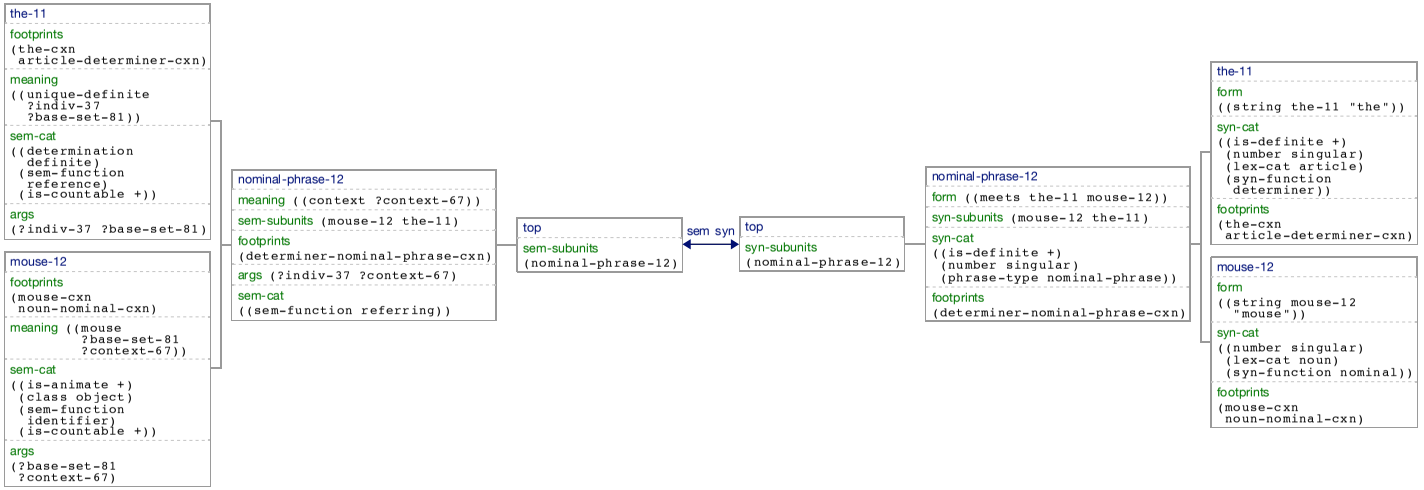

Figure 3. *Graphical display of transient structure when parsing or producing "the mouse". Each box represents a unit with its name and feature values. All features of the semantic poles are displayed on the left side and all features of the syntactic poles on the right side. Both poles are shown in more detail in Figure 2.*

FCG-symbols, such as names of units, names of features, names of syntactic categories and their values should all be chosen so as to make sense for a human reader. Rather than calling a unit `unit-1` it is better to call it `mouse` or `mouse-15`, if this unit was formed on the basis of the word "mouse". Additionally, rather than calling a syntactic category `G` and its value `M`, it is better to call them respectively `gender` and `masculine`. Indices (as 15 in `mouse-15`) are frequently used when many symbols that have similar roles are needed. The index has no particular meaning except to differentiate the symbol from another one such as `mouse-32` or `mouse-55`. The FCG-interpreter makes many symbols itself in the course of constructional processing, and it uses these indices abundantly.

Even simple grammars involve thousands of names and feature structures quickly become totally incomprehensible if abbreviations or irrelevant names are chosen. Nevertheless, the meaning of all these symbols only comes from the function of the named element in the overall system. It is not because a unit is called "`nominal-phrase`" that it starts functioning as a nominal-phrase or because the case of a noun is called "`nominative`" that the FCG-interpreter knows what nominative means. The role of an element in a feature structure is solely determined by the context in which it appears and what operations are carried out over it. A unit becomes a nominal-phrase because it has subunits categorized as article, adjective or noun, because it implies certain syntactic constraints among these subunits, like ordering or agreement, because it contributes in a particular way to reconstruct meaning, and so on.

2.2. List notation

There is also a list-notation of feature structures, used for typing feature structures through an editor or for looking at feature structures which are simply too big to display graphically. The list-notation is of the form

(*semantic-poles* `<-->` *syntactic-poles*)

where both the semantic poles and the syntactic poles consist of the feature structures listing the semantic respectively syntactic features of each unit. Each unit in list notation has a name followed by an (unordered) list of features and their values:

```
(unit-name
  (feature1 value1)
   ...
  (featuren valuen))
```

The value can either be a single item or a set of items. The ordering in which the features are listed is insignificant. The value of a feature may itself consist of a list of features and values. For example, the value of the `sem-cat` feature in the example below is:

```
((determination definite)
 (sem-function reference)
 (is-countable +))
```

This value consists of three sub-features (`determination`, `sem-function` and `is-countable`) with each their own respective (single) value. The ordering of these sub-features in the list is irrelevant. The features that directly depend from a unit are called *unit-features* (such as `meaning` or `args` in the semantic pole of `nominal-phrase-12`). The others are called *sub-features*.

An example of a list-notation for the same feature structure as shown in Figure 3 follows. The unit names are in bold and the unit features in italics. The features and values occurring in this example are all explained further on. In the semantic pole, there is a unit for `top`, which has one semantic subunit called `nominal-phrase-12`. `nominal-phrase-12` has two semantic subunits: `mouse-12` and `the-11`. The same unit-names are found on the syntactic pole with pending syntactic features.

```
( (top
   (sem-subunits (nominal-phrase-12)))
  (nominal-phrase-12
   (sem-subunits (mouse-12 the-11))
   (meaning ((context ?context-67)))
   (args (?indiv-37 ?context-67))
   (sem-cat ((sem-function referring)))
   (footprints (determiner-nominal-phrase-cxn)))
  (the-11
   (meaning ((unique-definite ?indiv-37 ?base-set-81)))
   (args (?indiv-37 ?base-set-81))
   (sem-cat
     ((determination definite)
      (sem-function reference) (is-countable +)))
   (footprints (the-cxn article-determiner-cxn)))
  (mouse-12
   (meaning ((mouse ?base-set-81 ?context-67)))
   (args (?base-set-81 ?context-67))
   (footprints (mouse-cxn noun-nominal-cxn))
```

```
   (sem-cat
     ((is-animate +) (class object)
      (sem-function identifier) (is-countable +)))))
<-->
( (top
   (syn-subunits (nominal-phrase-12)))
  (nominal-phrase-12
   (syn-subunits (mouse-12 the-11))
   (form ((meets the-11 mouse-12)))
   (syn-cat
     ((is-definite +) (number singular)
      (phrase-type nominal-phrase)))
   (footprints (determiner-nominal-phrase-cxn)))
  (the-11
   (form ((string the-11 "the")))
   (syn-cat
     ((is-definite +) (number singular)
      (lex-cat article) (syn-function determiner)))
   (footprints (the-cxn article-determiner-cxn)))
  (mouse-12
   (form ((string mouse-12 "mouse")))
   (syn-cat
     ((number singular) (lex-cat noun)
      (syn-function nominal)))
   (footprints (mouse-cxn noun-nominal-cxn))))
```

The feature structures used in FCG do not fundamentally differ from those used in other feature-structure based formalisms. For example, a more traditional representation of the syntactic pole of the transient structure starting from `nominal-phrase-12` (leaving out the footprints) is shown in Figure 2.2.

The hierarchy is represented here by embedding one feature structure into another one. Units do not have names because structure sharing is used instead. In FCG, each unit is given a name and these names are used when explicitly defining the subunits of a unit. This has the advantage that new subunits can be added or units can be moved in the tree simply by changing the value of the subunits feature. The advantage of using List-notation is that editors adapted to symbolic programming (such as EMACS) come with all the necessary facilities for efficiently editing list structures.

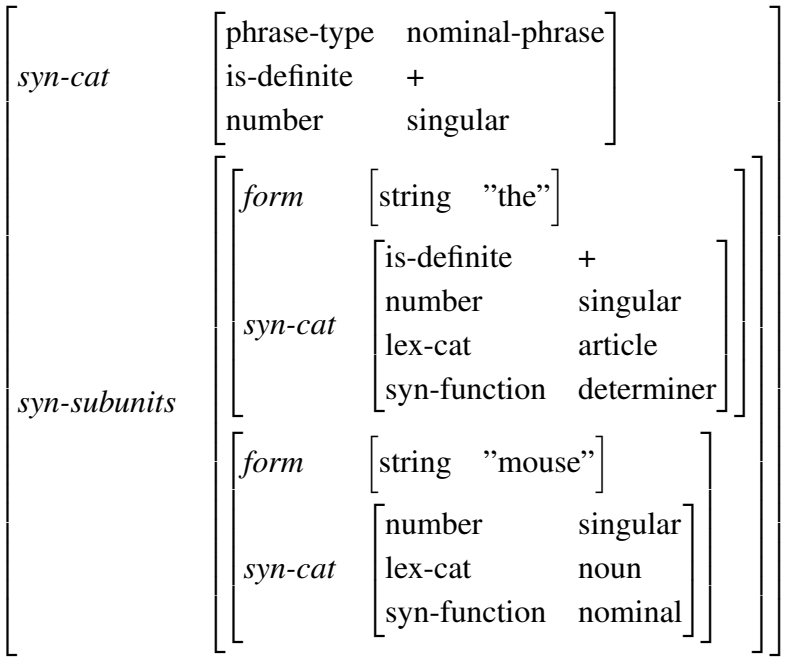

Figure 4. *A more traditional representation of feature structures.*

The set of possible unit features and their values is entirely open. The linguist may introduce new ones by just using them. They do not need to be declared in advance. A core set of basic unit features has nevertheless become standard practice and it is advisable to use them, but the values of these features typically vary substantially from one grammar to another. For example, a grammar for Russian aspect would need to represent all sorts of aspect distinctions which are entirely absent from a grammar for Japanese. The remaining subsections describe the main unit features in more detail.

2.3. Representing Hierarchy

The `sem-subunits` and `syn-subunits` features are used to represent the hierarchy of units and subunits in the semantic pole and the syntactic pole respectively, and they are filled by an unordered list of names of subunits. By distinguishing between semantic and syntactic subunits, it is possible that the hierarchical structure

on the semantic side is different from that on the syntactic side. For example, grammatical function words like "by" in passive constructions would not have a separate unit in the semantic pole, and there are possibly units on the semantic pole which do not show up explicitly in the syntactic structure or complete form of the sentence.

An example of sem-subunits and syn-subunits is seen in the nominal-phrase unit in Figure 3. The `syn-subunits` of `nominal-phrase-12` are `the-11` and `mouse-12`, written in list-notation as:

```
(syn-subunits (the-11 mouse-12))
```

In graphical representations of feature structures, such as in the one shown in Figure 2, the subunit features are used to draw the hierarchical structure.

The list of units of the `sem-subunits` and `syn-subunits` features is considered to be unordered. Thus, an equivalent way to specify the syn-subunits of `nominal-phrase-12` is:

```
(syn-subunits (mouse-12 the-11))
```

If ordering constraints need to be imposed on units, at whatever level of the hierarchy, they have to be represented explicitly as part of the form constraints of the relevant unit using predicates like `preceeds` or `meets`, as in other constraint-based formalisms such as GPSG (Gazdar et al., 1985). (See the next subsection). The explicit representation of ordering makes it possible to handle languages with no strict or much freer word order, without having recourse to movement-based approaches to scrambling (Fanselow, 2001). When there is no order imposed by the grammar, no order needs to be represented and the process rendering a feature structure into an utterance will make random decisions. Moreover decisions on order can be progressively refined in language production as more information becomes available. For example, it could be that the ordering of constituents inside nominal phrases is already known but their ordering in the sentence as a whole still remains to be decided.

2.4. Representing the Form of an Utterance

The `form` feature on the syntactic pole contains a description of the form characteristics contributed by the unit. They are expressed in terms of predicates over units. For example, the predicate `string` specifies which string is associated with a unit, as in

```
(string mouse-12 ‘‘mouse’’)
```

which states that the unit `mouse-12` introduces the string "mouse". The predicate `meets` specifies which units immediately follow each other in the utterance, as in

```
(meets the-11 mouse-12).
```

Form constraints can also be expressed over hierarchical units. For example, it is possible to express that the unit `nominal-phrase-43` has to follow the unit `verb-phrase-4` by saying

```
(preceeds verb-phrase-4 nominal-phrase-43)
```

Because the form of utterances is expressed with predicates, any kind of property can be included in principle. It is easy to add information about intonation, tone, or stress patterns by introducing predicates that specify these properties. The constraints could even include properties of gestures or pauses in speech.

The FCG-system includes a `render`-component that collects all form predicates for all units at each level of the hierarchy and turns them into a linearly-ordered utterance. It also includes a `de-render`-component that takes an utterance and turns it into a list of units with the relevant form characteristics represented explicitly using predicates. New form predicates can be introduced easily by extending these render and de-render operations.

The *complete form* of a unit is defined as the union of all the values of the forms of all its subunits plus its own form, and is computed dynamically whenever needed. Thus, for the unit `nominal-phrase-12` in the example above, the complete form is equal to:

```
((meets the-11 mouse-12)
 (string the-11 "the")
 (string mouse-12 "mouse"))
```

This set of expressions describes the complete (written) form of the phrase “the mouse”.

2.5. Representing Meaning

In principle, the representations used for the meaning of a unit are open-ended as well. The linguist may for example decide to use a logic-based representation within the tradition of formal semantics, such as Minimal Recursion Semantics (Copestake et al., 2006) which has its roots in Montague semantics, or use some kind of frame semantics as explored by many cognitive linguists (Baker et al., 1998). FCG is part of a larger project that uses embodied cognitive semantics, but details of the representations and mechanisms used for this are beyond the scope of the present chapter (see (Steels, 2012) for more information).

In the examples discussed further in this paper, a logic-style representation is used, that is based on predicates and arguments. The `meaning` feature on the semantic pole contains the predicates contributed by the unit and the `args` feature contains those arguments of these predicates that can be used to combine the meaning of this unit with meanings contributed by other units. For the noun "mouse" (semantic pole of unit `mouse-11`) the meaning is equal to the following expression:

```
((mouse ?base-set-81 ?context-67))
```

The numbers for these variables (as well as for names of other units) are generated automatically by the FCG-interpreter or semantic processes that formulate the meaning that needs to be conveyed. Mouse is a predicate which delineates the set of mice `?base-set-81` as a subset of the context `?context-67`. Both arguments `?base-set-81` and `?context-67` can be used in further combinations and are therefore listed in the `args` feature. `unique-definite` has two arguments an individual and a base-set. It checks or establishes that its base-set has only one member and it is equal to the individual.

The meaning of the utterance as a whole is distributed over the different units in the feature structure. The *complete meaning* of a unit is defined as the union of the values of the meanings of all its subunits plus its own, as in Copestake et al. (2006). It is again computed dynamically whenever needed. For the example above, the complete meaning of unit `nominal-phrase-12` is equal to the following set of predicate argument expressions:

```
((context ?context-67)
 (unique-definite ?indiv-37 ?base-set-81)
 (mouse ?base-set-81 ?context-67))
```

2.6. Representing Categorizations

Grammar uses categorizations to establish abstract associations between meaning and form (as illustrated in Figure 5). These categorizations typically take the form of feature-value pairs, such as `(number singular)` or `(gender masculine)`, sometimes with binary values, such as `(definite +)` or `(is-animate -)`. Categorizations may also take the form of a single predicate or even relations. The possible categorizations usable in FCG are entirely open. Categorizations are values of unit-features that indicate what the categorization is about: `prag-cat`, `sem-cat`, `syn-cat`, `phon-cat`. Each of these features has a list of categorizations as its value. The ordering in which they are specified does not matter.

For example, the `syn-cat` in the mouse-unit in Figure 2 includes three syntactic categorizations: the lexical category `lex-cat` with value `noun`, the category `number` with value `singular`, and the category `syn-function` with value `nominal`.

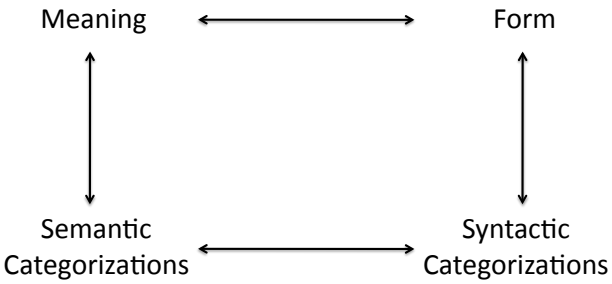

Figure 5. *The grammar square depicts the different associations between meaning and form that constructions establish. Meaning can be directly related to form, as in the case of words, or it is expressed through the intermediary of semantic and syntactic categorizations.*

Because the set of possible categorizations used in a grammar is entirely open, it is the task of the linguist to identify which ones are necessary and sufficient to handle the phenomena in the language being investigated. FCG does not propose an a priori set of universal categorizations, partly because no consensus on this matter exists among linguists, perhaps because there does not appear to be a universal a priori set (Haspelmath, 2007).

It is useful to divide categorizations along the lines of traditional levels of analysis to improve the readability of transient structures. So distinctions are made between the following unit features on the semantic pole:

1. `sem-cat`: This unit feature contains all the semantic categorizations. These are reconceptualizations of the meaning to be expressed, such as the abstract roles of participants in events (agent, patient, beneficiary, etc.), object distinctions like countability (count vs. mass), semantic functions (reference, qualifier, modifier), etc.

2. `prag-cat`: This unit feature contains all the pragmatic categorizations. They are related to discourse functions of units, for example whether they are viewed as the topic of the sentence, part of the foreground or background, and so on.

The following unit features are used to group the categorizations on the syntactic pole:

1. `syn-cat`: This feature groups all the syntactic categorizations, such as the part of speech of the unit (called the lexical category), syntactic features such as number, definiteness, (syntactic) case, syntactic function, etc.

2. `phon-cat`: This feature groups categorizations that are relevant to the morphology, phonology and phonetics of a unit in as far they play a role in grammar, such as whether the stem is regular or irregular, ends on a consonant cluster, has a rounded vowel in the stem, etc.

The distinction between these different types of categorizations is not always clear-cut and is up to the grammar designer anyway. For example, the distinction between count and mass noun is often seen as a syntactic categorization but it could just as well be considered a semantic categorization, as was done in the example of "the mouse" shown in Figure 2. It often does not much matter where a categorization is placed, because semantic and syntactic processing always go hand-in-hand in FCG. As a general heuristic, categories that have a clear semantic basis are considered semantic and those that are clearly conventional are considered syntactic. For example, 'agent' is clearly a semantic categorization which is grounded in the way the role of a participant in an event is construed whereas 'nominative' is clearly a syntactic categorization, because, even though agents are usually nominative, other semantic roles could also be mapped into a nominative, as in passive constructions.

3. Constructional Processing

Before discussing the representation of constructions in more detail, it is useful to take a first look at how constructions are applied in language processing. A simple nominal phrase is taken as the example, using the lexical and phrasal constructions described in more detail later in this book (Steels, 2011).

To begin, parsing starts with a unit, by definition called `top`, that contains all the information that could be gathered about the utterance by lower level speech recognition processes: the strings, the intonation and stress patterns, the linear ordering of strings, etc. For the phrase "the mouse", the top would initially at least contain the information that there are two word strings: "the" and "mouse", and that they strictly follow each other. (See Figure 6.) The semantic pole of this unit here is empty, but it could already contain partial meanings that are anticipated by the listener based on prior discourse.

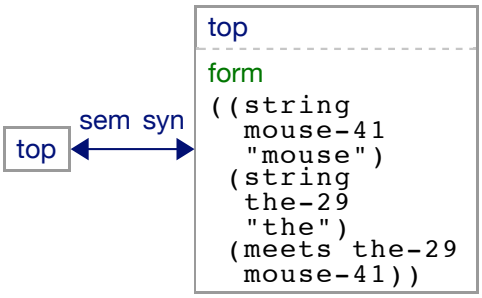

Figure 6. *The initial top-unit contains in the syntactic pole all information that could be extracted about the form of the utterance.*

The top acts as a kind of input buffer from which information is gradually taken by constructions to build the transient structure. It is assumed in this paper that this input buffer is filled with all the information provided by a complete utterance at once, but of course in a more realistic setting it would be filled gradually as the individual words sequentially come in. The input buffer is represented as a unit at the top of the hierarchy of syntactic and semantic subunits so that all the machinery used for applying constructions can be used both for the top unit and all its descendants.

A construction is a kind of daemon on the look out for whether it can expand a transient structure, and, if finds something, it performs the expansion. The first step (looking out) involves an operation called *matching* and the second step (expansion) involves an operation called *merging* (see Steels & De Beule, 2006, for a more formal definition of matching and merging in FCG). More precisely, *matching* means that one pole of a construction is compared with the corresponding pole in the target (the transient structure being expanded) to see whether there are non-conflicting correspondents for every unit, feature, and value. Which pole is chosen depends on whether parsing or production is going on. In the case of parsing, the syntactic poles are compared. In the case of production, the semantic poles are compared.

For example, if the pattern is equal to

```
(noun-unit (syn-cat ((lex-cat noun))))
```

and the target is equal to

```
(noun-unit (syn-cat ((lex-cat noun))))
```

then both structures match. But if the target were equal to

```
(adjective-unit (syn-cat ((lex-cat adjective))))
```

the pattern would not match because the name of the units are not the same (`noun-unit` versus `adjective-unit`) and the value for `lex-cat` is different as well (`noun` versus `adjective`). The ordering of subunits or of syn-cat and sem-cat values does not matter, so that

```
((number singular) (case nominative))
```

matches with

`((case nominative) (number singular))`. Note that in FCG the names of units play no role whatsoever. We could use the name `A-U` everywhere where we use `adjective-unit` and it would not change the behavior of the system. The name `adjective-unit` is only used because it makes it easier for us to read feature structures.

Merging means that those parts of the other pole of a construction (the one that was not used in matching) which are missing in the corresponding pole in the target are added *unless they are in conflict* in which case the whole operation fails. Merging performs first a matching process to find out what structures are already there and whether there is any conflict. For example, if the pattern P is

```
P = ((noun-unit
      (syn-cat (lex-cat noun)
               (number singular))))
```

and the target T is

```
T = ((noun-unit (syn-cat (lex-cat noun))))
```

then the result of merging `M` = merge(`P`, `T`) would be

```
M = ((noun-unit
      (syn-cat (lex-cat noun)
               (number singular))))
```

The category `number` with value `singular` has been added to `T`. Merging fails if there are conflicting correspondents.

Distinguishing matching and merging gives more control over the way a construction is applied and thus makes the definition of constructions easier. It has also been a key innovation for achieving reversibility, because the semantic and syntactic poles of a construction take on different roles in parsing and production:

1. When producing, the semantic pole of a construction is matched against the semantic pole of the transient structure, and, if a match succeeds, the syntactic pole of the construction is merged with the syntactic pole of the transient structure. Because merging is blocked when there are conflicts, the syntactic

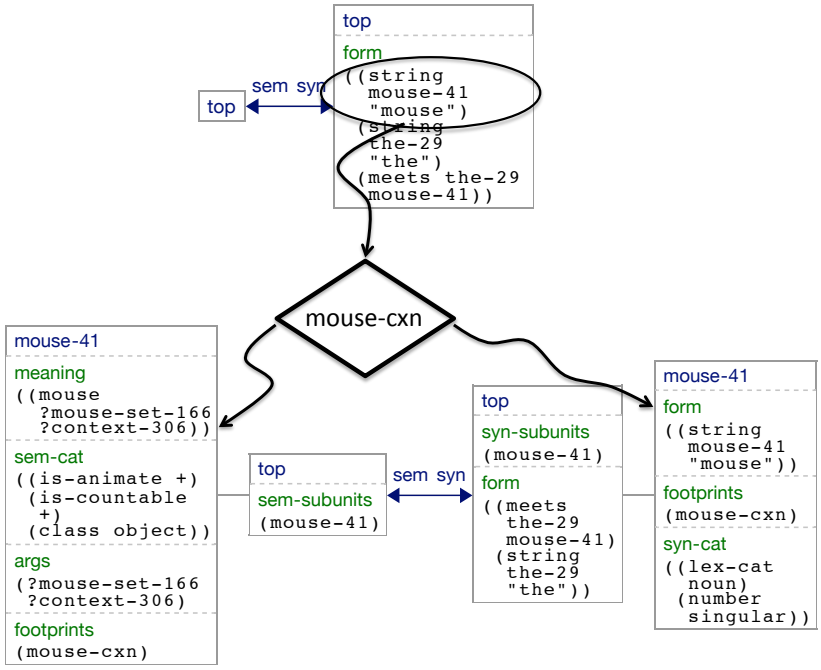

Figure 7. *A lexical construction handling the word "mouse" has created a new sub-unit hanging from the top. It contains in the syntactic pole all additional information that the construction provides, such as the number and lexical category of the word and, in the semantic pole, information relevant to meaning, such as the fact that "mouse" introduces an animate entity.*

pole of a construction can still prevent application. Nevertheless, the semantic pole is the first hurdle before the rest of the construction is examined.

2. When parsing, the syntactic pole of a construction is matched against the syntactic pole of the transient structure, and, if that succeeds, the semantic pole of the construction is merged with its semantic pole. Because merging still tests for conflicts, the semantic pole can prevent application as well, but it is the syntactic pole which first specifies what constraints need to be satisfied in order to even start considering a construction.

To return to the parsing process, lexical constructions associate meanings directly with lexical material (words, morphemes). They examine the top, and, if there are certain form elements that they can cover, for example they notice a certain word-string (in the matching phase), they become active and add more information to the transient structure built so far (in the merge phase). Typically for lexical constructions this process includes creating a new unit, hanging it from the top, encapsulating the information covered in the top by putting it into the new unit, and adding some more information to that.

Thus there could be a lexical construction that sees the string "mouse", creates a new sub-unit for it (here called `mouse-41`), moves the string "mouse" from the `top` to this `mouse-41` unit, and adds further syntactic categorizations to the unit, namely that the lexical category (part of speech) is noun and the number singular. (See Figure 7.) Information is added to the new unit's semantic pole as well, namely that it introduces the predicate `mouse` with two arguments `?mouse-set-166` and `?context-306`, and that its semantic categorizations include being animate and countable. The semantic and syntactic pole are always displayed separately so the newly constructed unit appears twice in the graphical representation.

Other lexical constructions do the same sort of thing: They are on the look out for whether the forms they can cover appear in the input, and, if so, they create more units and add information about their syntactic and semantic properties. The ordering of construction applications does not entirely matter. Constructions are (conceptually) applied in parallel, and a construction triggers as soon as its conditions are satisfied. However, sometimes ordering constraints are imposed to speed up language processing or to handle phenomena like unmarked forms.

The application of constructions is recursive in the sense that constructions can trigger on information contained in the units created by other constructions. There might be a `determiner-nominal-construction` that looks at the unit created based on the word "the" and the unit created for the word "mouse" in order to check whether they occur in a particular ordering, have the same values for number (here singular) and satisfy other semantic constraints (such as countability). If that is the case, this construction would build a new unit combining `the-29` and `mouse-41` and reorganize the overall structure by encapsulating them as daughter nodes, as shown in Figure 8. The determiner-nominal construction would not only build syntactic structures. It would also build additional parts of the meaning that follow from this combination, for example, that one of the arguments of the article is the same as one used by the nominal (i.c. `?mouse-set-166`). The construction would also

percolate semantic categorizations from the constituents to the phrasal parent and possibly add some new categorizations of its own.

Construction application goes on until no more constructions can be applied. The meanings contained in all the meaning features of all units in the final transient structure are collected and interpreted within the present discourse context. Normally, the top, acting as input buffer, becomes gradually empty as the form constraints are moved to units lower in the hierarchical structure. When form constraints are left over, for example a string is left in the top, it indicates that there were some aspects of the utterance which could not be treated, perhaps because there is an unknown word.

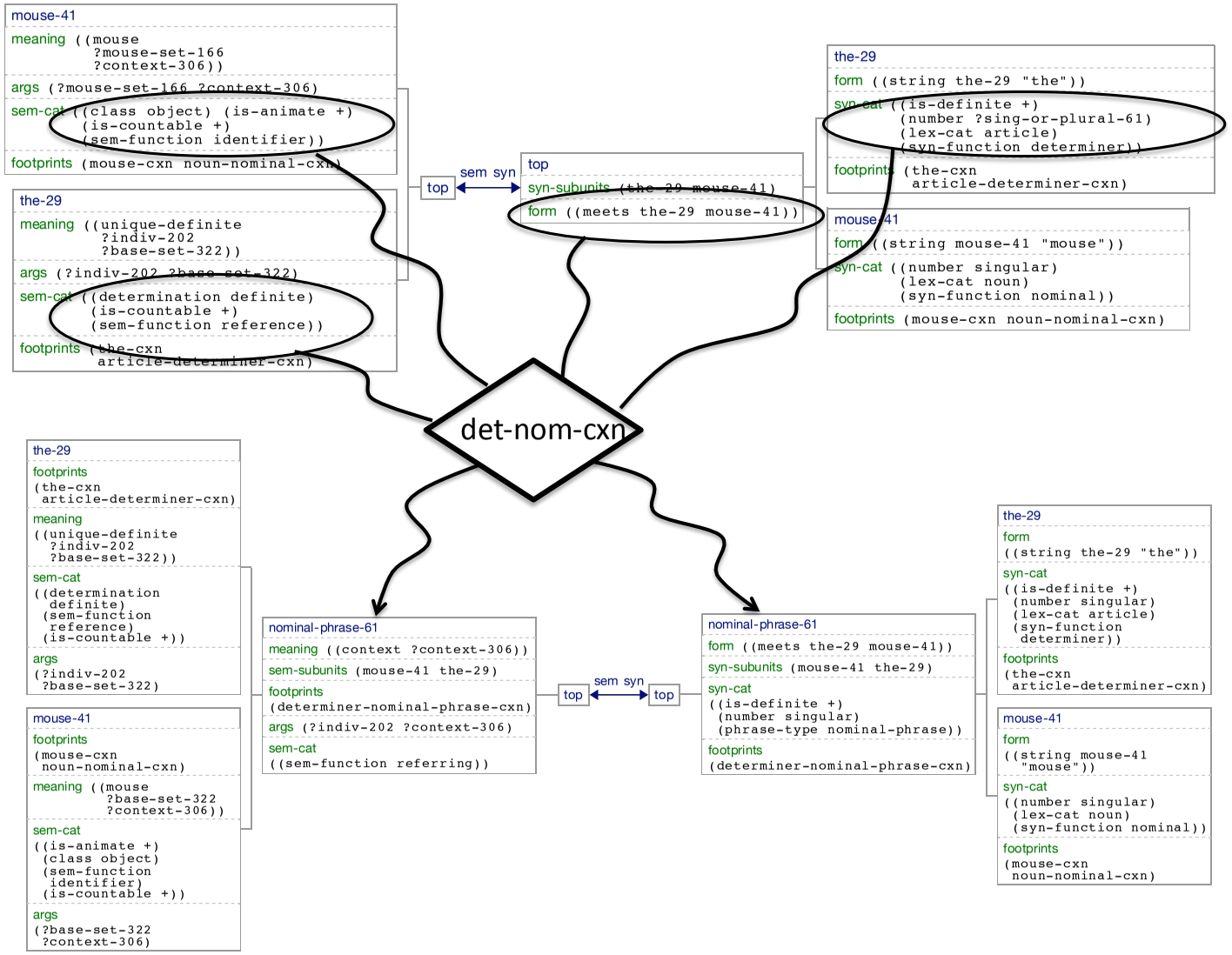

Figure 8. *Top: transient structure after application of lexical constructions for "the" and "mouse". Bottom: transient structure after the determiner-nominal construction has applied. This construction checks for various syntactic and semantic properties and then builds a nominal phrase with the units for "the" and "mouse" as subunits.*

FCG does not see the task of language processing as checking whether a sentence is grammatical. The goal is to come up with the best possible interpretation given the input and all available constructions. It is quite common in natural dialogue that some words are unknown or not well recognized, or that a phrase has not been built correctly by the speaker or a construction has been stretched to the limit. Consequently there are often gaps and difficulties to build a complete well-formed structure. Nevertheless, listeners can usually interpret such sentences by relying on the context and tacit background knowledge.

The production process runs in an entirely analogous way, except that the top now contains all information about the meaning that the speaker wants to express. Therefore, it acts again as an input buffer, and, if meanings are left in the semantic pole of the top at the end of processing, it indicates the presence of a problem: possibly words were missing to express some meanings, or grammatical constructions were missing or could not be applied. Constructions still apply by a process of matching and merging, but the poles are reversed:

1. Matching: The semantic pole of the construction is matched against the transient structure built so far.

2. Merging: The match is successful, and information from the syntactic pole is merged in with the transient structure built so far.

Usually lexical constructions trigger first. They are on the look out for the presence of certain meanings, and, if they find them, they create new subunits which encapsulate these meanings and add syntactic and semantic information to them. The lexicon is thus primary, both in parsing and production. If the speaker does not know any grammar but has already an adequate lexicon, an utterance could already be produced by rendering the bare words in a random order.

Grammatical constructions build further on the lexical units, covering additional aspects of meaning that are expressed grammatically and integrating information already contained in other units. For example, the `determiner-nominal-construction` triggers if it finds units for a determiner and nominal that satisfy specific semantic and syntactic properties, and it then adds more form constraints, namely that the constituents have to follow each other in the final sentence. Thus, constraints on the form of an utterance are progressively collected, based on the meaning that needs to be expressed and the rest of the structure built so far. This process goes on until no more constructions can be applied. At that point there is hopefully enough information available in the syntactic poles of

all the units so that a concrete utterance can be constructed. Of course it may be possible that certain constructions are missing, but there could nevertheless still be enough information to produce sentence fragments that might be understood by an intelligent listener.

4. Representing Constructions

In this section we begin to look in more detail at the way constructions are defined in FCG. The representational tools that we have seen already for transient structures are extended to be able to define constructions, which means that they are made more abstract using variables, special operators, and ways to deal with hierarchical structure.

4.1. Basics

A construction has fundamentally the same structure as a transient structure. It contains a set of units and features associated with each of these units. They are decomposed into a semantic and a syntactic pole and all the conventions introduced earlier for transient structures apply, including which unit-features may appear (`syn-subunits`, `form`, `syn-cat`, etc.) and what their possible values may be.

There is a general function called `def-cxn` used in the following way to define a construction:

```
(def-cxn name parameters
 semantic-poles
 <-->
 syntactic-poles)
```

The *name* is the name of the construction. The *parameters* specify properties of constructions not discussed in this introductory chapter. Below is an example of a possible construction called `mouse-cxn`, which is in fact entirely similar to the transient structures discussed earlier:

```
(def-cxn mouse-cxn ()
 ((mouse-unit
   (meaning ((mouse mice-set context)))
   (args (mice-set context))
   (sem-cat
     ((is-animate +) (class object)
```

```
      (sem-function identifier) (is-countable +)))))
 <-->
 ((mouse-unit
   (form ((string mouse-unit "mouse")))
   (syn-cat
     ((number singular) (lex-cat noun)
      (syn-function nominal))))))
```

Although constructions strongly resemble transient structures, they must now be made more abstract to allow matching and merging with the relevant transient structures. Simply leaving out details creates this effect. For example, a di-transive construction defines a pattern of syntactic usage on the syntactic pole but says almost nothing about the internals of the constituents involved, except that they are nominal phrases. FCG has two additional mechanisms for achieving abstraction, the first of which is based on the use of variables. The second consists of operators that allow a more refined way to specify how feature values have to match and how they have to merge.

4.2. Variables in FCG

FCG uses logic variables for addressing a part of a structure, whether this is a unit name, a feature-value pair, a set of units, or the value of a feature. FCG-variables are denoted by putting a question-mark in front of the variable name, as in `?unit` or `?tense`. The `mouse-cxn` can thus be made more abstract by using variables for the unit-names, so that the construction will match with any unit that matches with its features, and variables for the arguments in the meaning. (All items made variable are in bold.)

```
(def-cxn mouse-cxn ()
 ((?mouse-unit
   (meaning ((mouse ?mice-set ?context)))
   (args ( ?mice-set ?context))
   (sem-cat
     ((is-animate +) (class object)
      (sem-function identifier) (is-countable +)))))
 <-->
 (( ?mouse-unit
   (form ((string ?mouse-unit "mouse")))
   (syn-cat
```

```
((number singular) (lex-cat noun)
 (syn-function nominal))))))
```

An FCG-variable functions like a variable in mathematics or computer programming. It can become bound to a particular value and later be used to refer to this value. The binding is not based on an explicit assignment but occurs as a side effect of the matching process:

> When two structures are matched and an unbound variable is encountered in the pattern or target, then this variable gets bound to the element occurring in the same position in the pattern resp. target. If the variable is already bound, then the element in the same position must be equal to the binding of the variable. If both are variables, then these variables are considered to be equal, making them equal for the rest of the matching process.

For example, if the pattern is equal to `(gender ?gender)` and the target structure is equal to `(gender feminine)`, then after matching these two structures, `?gender` has become bound to `feminine`. If the same variable occurs somewhere later in the pattern it is considered to be equal to `feminine`. Variables may be bound to other variables. For example, if the pattern is equal to `(gender ?gender)` and the target structure equal to `(gender ?unknown-gender)`, then after matching these two structures, `?gender` has become bound to `?unknown-gender`. Further on, if any one of these two gets bound to a value, the other one is bound as well.

The behavior of variables in merging is as follows:

> When two structures are merged and a variable is encountered in the pattern or target, then this variable is replaced by its binding. When a variable is unbound, the variable is left as is. When a variable is bound to another variable, then both are replaced by the same new variable.

For example, suppose that `?gender` is bound to `feminine`, then if `(gender ?gender)` occurs in the transient structure being merged, it will appear as `(gender feminine)`. The merging of two structures may give rise to new variable bindings which are then used in merging. For example, if `(gender feminine)` occurs in the target structure and `(gender ?gender)` in the pattern, then `?gender` will become bound to feminine as part of the merging process, if this variable occurs later again, `?gender` will considered to have been bound to `feminine`.

Variables may occur anywhere in constructions as well as in transient structures. Their most obvious role is to act as slots that are filled in the matching phase by

information from a transient structure and instantiated in the merge phase. For example, the unit names in constructions are always variables so that they can get bound to those units in a transient structure whose features and values match with the ones in the construction. The conditional pole (the semantic pole in production or the syntactic pole in parsing) establishes bindings, which are then used to retrieve their correspondents in the contributing pole (the syntactic pole in production and the semantic pole in parsing).

Bindings can come from either one of the two structures being matched or merged, just as in logic-style unification. The FCG-interpreter takes care that variables of pattern and target do not get confused, by renaming all variables in feature structures before they get matched and merged. It is entirely possible to bind one variable to another variable, producing the consequence that their future bindings must be equal. For example, when matching the structure

`S = (syn-subunits (?adjective ?noun))`

with

`T = (syn-subunits (adjective-brown ?some-noun))`,

the variable `?adjective` gets bound to `adjective-brown`, and the variable `?noun` gets bound to another variable `?some-noun` . When these two structures are merged it produces

`M = (adjective-brown ?noun-var)`

The bound variable is replaced by its binding and a single (new) variable (here called `?noun-var`) has been introduced to replace the two variables that were equal.

The possibility that variables can be bound to other variables has many utilities. For example, it can be used to link together meanings contributed by individual units. Suppose the adjective `brown` contributes the meaning `(brown ?referent-1)` and the noun `table` contributes the meaning `(table ?referent-2)` then the adjectival-nominal construction can link these meanings together by binding `?referent-1` to `?referent-2`. During merging, the variables are then replaced by a single new variable, as in

`((brown ?referent-3) (table ?referent-3))`.

This example illustrates that transient structures may also contain variables, which is not only useful for handling compositional semantics but also for leaving certain syntactic or semantic categorizations unspecified until enough information is available to make a decision. For example, it is possible in language production to enforce an agreement constraint between a determiner and a nominal for number without knowing yet what the value for number is going to be, simply by using the same variable for the value of the category in the units for the article and the noun.

See van Trijp (2011) later in this book for illustrations on how this functionality can be used for handling complex agreement phenomena.

4.3. Operators

Using variables is one way in which constructions can become more abstract. Another is namely by specifying in more detail which aspects of a feature-value pair have to match and how, so that some parts can be ignored or so that possible conflicts can be more clearly identified. Typically, matching has to be complete and precise, including the ordering of the elements, but the following operators can override that:

1. (== $element_1$ ... $element_n$): The *includes-operator* specifies that the value in the target should include the elements $element_1$, ... , $element_n$, but the target can contain more elements and the ordering no longer matters.

2. (==1 $element_1$ ...): The *uniquely-includes-operator* specifies that each of the elements should occur in the target, which may still include more elements, but that there should only be one value for the same feature. Again the ordering of the elements no longer matters. This information not only helps the matcher by avoiding consideration of unnecessary additional hypotheses, it also impacts merging, because without this operator the additional category-value pair would simply be added even if another value already is present.

3. (==0 $element_1$ ... $element_n$): The negation (or *excludes-operator*) specifies that the elements should *not* occur in the target. In this way, a construction is able to check for example whether the transient structure already contains a footprint left by the same or another construction that should prevent its further application.

Using these elements, the "mouse" construction shown earlier can now be made even more abstract (changes are in bold):

```
(def-cxn mouse-cxn ()
 ((?mouse-unit
   (meaning (== (mouse ?mice-set ?context)))
   (args (?mice-set ?context))
   (sem-cat
     (==1 (is-animate +) (class object)
      (sem-function identifier) (is-countable +)))))
```

```
<-->
((?mouse-unit
  (form (== (string mouse-unit "mouse")))
  (syn-cat
    (==1 (number singular) (lex-cat noun)
     (syn-function nominal))))))
```

The values of `meaning` and `form` use the includes-operator **==**, because the meaning must include the expression `(mouse ?mice-set ?context)` but may contain other expressions, and the form must include the string "mouse" but perhaps other form constraints. For example, if the utterance "the black mouse" is being processed the top-unit will contain also information that there is a string "the" and a string "black" and that there is an ordering constraint between them. `mouse-cxn` is however only interested in the string "mouse". The semantic and syntactic categorizations use the *uniquely-includes* operator **==1**, because each of their elements should have specific single values. `Number` can only be `singular`, `lex-cat` can only be `noun`, `syn-function` can only be `nominal`.

4.4. Hierarchical Structure

Feature structures, matching and merging, and logic variables are all quite standard representational mechanisms in Artificial Intelligence and Computational Linguistics, and algorithms for implementing them efficiently can be found in common textbooks (see for example Norvig (1992)). FCG packages these mechanisms in a way that they are adapted to represent and process complex transient structures and constructions. What is undoubtedly less common is the way FCG builds and manipulates hierarchical structures, which is done with a single powerful structure-building operator, known as the *J-operator*.

The J-operator has three arguments and performs two functions. The arguments are: a *daughter-unit*, a *parent-unit*, and possibly a set of *pending-subunits*. These are either specified with concrete names or with variables that have been bound elsewhere in the matching or merging process. When the daughter-unit is an unbound variable at the time of merging, a new unit will be created for it.

The first function of the J-operator is to hang the daughter-unit from the parent-unit by changing the value of the `syn-subunits` or `sem-subunits` slot of the parent-unit. If there are pending-subunits, these will then be attached from the daughter-unit. For example, a lexical construction typically must match in production with some part of the meaning to be expressed, and, if that is the case, the

construction creates a new unit containing the relevant word form in its syntactic pole and the meaning being covered in the semantic pole. The same construction performs a mirror operation in parsing, by matching with some word form observed in the input and then creating a new unit with a syntactic pole and a semantic pole.

Secondly, the J-operator can associate additional information with the daughter-unit. A lexical construction might want to associate syntactic categorizations with the new unit it created for the word stem (such as lexical categories, gender, number, etc.) as well as semantic categories (for example that the word introduces a motion-event or an inanimate object). If this information is already present, so much the better. If there is a conflict, the construction stops further application.

Units governed by the J-operator can occur in both poles of a construction. Consequently, a construction typically has a quadripartite structure. Each pole may contain units that are governed by a J-operator, further called *J-units*, and units that are required to be part of the existing structure, further called *conditional units*. These additions are illustrated with an expanded version of the lexical construction for "mouse" which now models the behavior sketched in section 3 (additions are in bold). The construction matches with meanings in production or forms in parsing which are present in the top-unit and it builds a new unit hanging from the top which contains the semantic and syntactic categorizations (as shown in Figure 7).

```
(def-cxn mouse-cxn ()
  ((?top-unit
     (meaning (== (mouse ?mice-set ?context))))
   ( (J ?mouse-unit ?top-unit)
     (args (?mice-set ?context))
     (sem-cat
       (==1 (is-animate +)
            (class object) (is-countable +)))))
 <-->
 ((?top-unit
   (form (== (string mouse-unit "mouse"))))
  ( (J ?mouse-unit ?top-unit)
    (syn-cat
      (==1 (number singular) (lex-cat noun))))))
```

Note that the name `?top-unit` is used here but this is only because often the construction operates with the top-unit of a transient structure. We could have used any other name for the variable that gets bound to `?top-unit`. Names of variables are chosen for readability but the name itself has no semantics attached to it.

The application of constructions in parsing or production can now be defined more clearly:

- When producing, the conditional units of the semantic pole of a construction are matched against their correspondents in the transient structure, but the J-units are ignored. If a match succeeds, the J-units of the semantic pole are merged with the semantic pole of the transient structure, followed by the syntactic pole of the construction merging with the syntactic pole of the transient structure (both the J-units and the conditional units).
- When parsing, the conditional units of the syntactic pole of a construction are matched against their correspondents in the syntactic pole of the transient structure, but the J-units are ignored. If a match succeeds, the J-units of the syntactic pole are merged with the syntactic pole of the transient structure, and all units of the semantic pole of the construction are merged with the semantic pole of the transient structure.

The process by which the mouse-construction gets applied begins with the following initial transient structure in parsing, with the semantic pole of the top still empty (See Figure 6):

```
((top))
 <-->
((top
  (form
    ((string mouse-41 "mouse")
     (string the-29 "the")
     (meets the-29 mouse-41)))))
```

Matching the syntactic pole of the mouse-construction with this transient structure yields the bindings `(?mouse-unit . mouse-41) (?top-unit . top)`. After merging, the syntactic pole of the transient structure is as follows:

```
((top
   (syn-subunits (mouse-41))
   (form
     ((string mouse-41 "mouse")
      (string the-29 "the")
      (meets the-29 mouse-41))))
 (mouse-41
   (syn-cat
     ((lex-cat noun) (number singular)))))
```

After the semantic pole of the construction is merged, the following semantic pole is created:

```
((top
   (sem-subunits (mouse-41))
   (meaning ((mouse ?base-set-322 ?context-306))))
 (mouse-41
   (args (?base-set-322 ?context-306))
   (sem-cat
     ((is-animate +) (class object)
      (is-countable +)))))
```

The meaning feature from the mouse-construction has been merged into the meaning feature of `top`, and args and sem-cat features have been added to the mouse-unit. This operation was illustrated in Figure 7.

The J-operator is made more versatile by introducing a way to *tag* parts of a feature structure so that they can be moved somewhere else in the hierarchy. Lexical constructions typically remove part of what they covered in the top-unit and encapsulate it in the unit they created, so that other constructions would not trigger to express this meaning or cover the same forms. FCG does this operation with a *tagging operator*. The tagging operator written as `tag` has two arguments: a variable, known as the *tag-variable*, and a set of features and values that will be bound to the tag-variable. The normal matching process is still used to check whether the features and values match. If a tag-variable re-occurs inside a unit governed by a J-operator, then the structure is *moved* from its old position to its new position.

This process is illustrated in the following further refinement of the mouse-construction (additions are in bold):

```
(def-cxn mouse-cxn ()
  ((?top-unit
     (tag ?meaning
          (meaning (== (mouse ?indiv)))))
   ((J ?mouse-unit ?top-unit)
     ?meaning
     (args (?mice-set ?context))
     (sem-cat
      (==1 (is-animate +) (class object)
           (is-countable +)))))
  <==>
  ((?top-unit
```

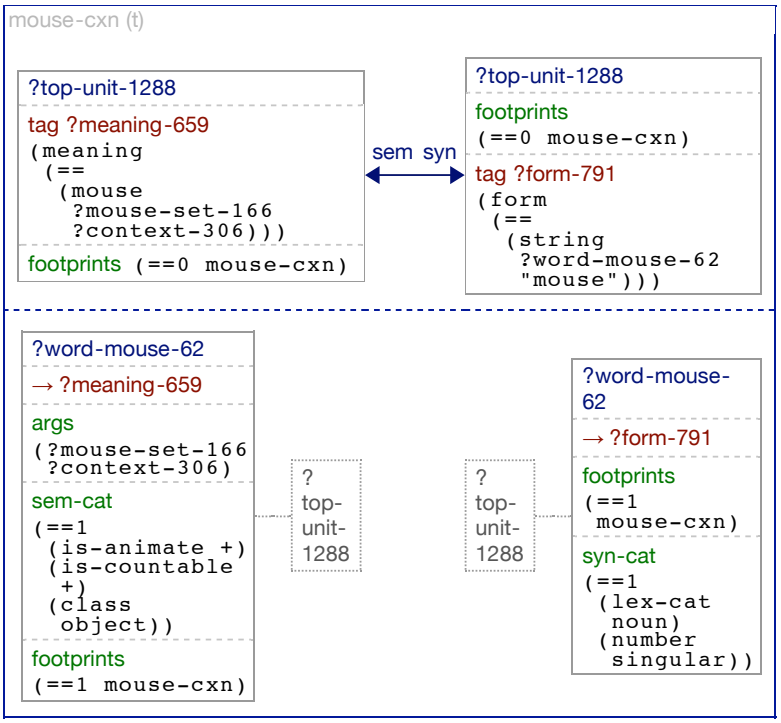

Figure 9. *Example of the graphical display of a construction. The top part shows the feature structure that the construction is looking for on the semantic and syntactic side. The bottom shows the units that are constructed by the J-operator. The tag variables are bound in the top part and used in the bottom part.*

(tag ?form
(form (== (string ?mouse-unit "mouse")))))

```
((J ?mouse-unit ?top-unit)
```
?form
```
(syn-cat
  (==1 (lex-cat noun) (number singular))))))
```

The meaning feature in the semantic pole of the top-unit gets bound to the tag `?meaning` and then moved into the mouse-unit. The form feature in the syntactic pole of the top gets bound to the tag `?form` and moved into the syntactic-pole of the mouse-unit, so that the transient structure after matching and merging becomes:

```
((top-unit-5
   (sem-subunits (mouse-41)))
 (mouse-41
   (args (?mice-set-2 ?context-306))
   (meaning ((mouse ?mice-set-2 ?context-306)))
   (sem-cat
     ((is-animate +) (class object)
      (is-countable +)))))
 <-->
((top
   (syn-subunits (mouse-41))
   (form
     ((string the-29 "the")
      (meets the-29 mouse-41))))
 (mouse-41
   (form ((string mouse-41 "mouse")))
   (syn-cat
     ((lex-cat noun) (number singular)))))
```

In this paper, several examples of graphical displays of transient structures have been shown. Constructions have also a graphical display, with an example given in Figure 9. The top half shows what the constructions expect to be present in the transient structure (the conditional units), and the bottom half shows what is added by the J-operator (the J-units).

The J-operator is a very powerful structure building operator. It can create new units, build and reorganize the hierarchical structure, and add information to existing or new units. It is the only structural operator used in FCG.

5. Influencing Construction Application

The application of constructions is seldom a simple matter because often more than one competing construction is applicable. Two constructions are competing if their triggering conditions are the same, either on the semantic pole in case of synonymy or on the syntactic pole in case of ambiguity or polysemy. To deal with this kind of competition, the FCG-interpreter must set up a *search space* to track the different chains of possible construction applications.

5.1. The Search Space

The constructions which operate sequentially on the same transient structure form a linear chain. When more than one construction can apply, the chain forks into different paths, and we get a search tree (see Figure 10). The search space is the set of all possible nodes in a tree. For example, suppose we have two lexical constructions for table, one defining table as a piece of furniture and another as an arrangement in rows and columns and we run these on the input "the table", then we get the search tree as shown in Figure 10.

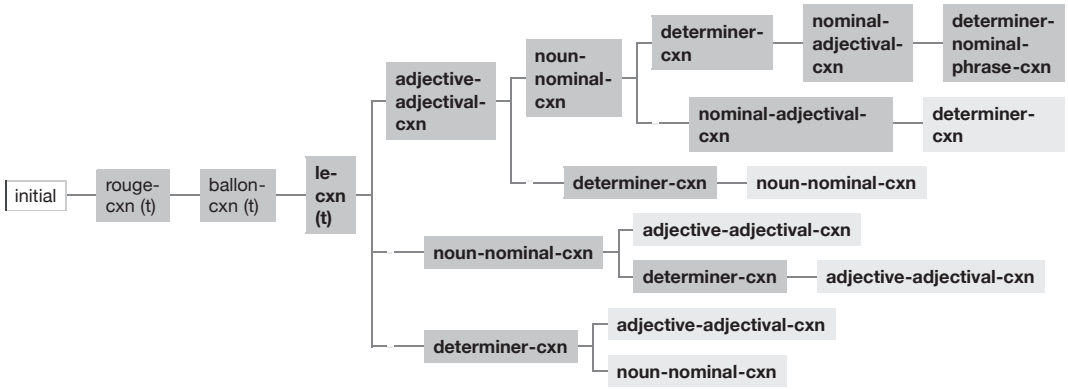

Figure 10. *Graphical representation of the search tree automatically created by the FCG-system. The linguist can browse through this tree and click on nodes to see which construction applied and what the state before and after application was. There are two chains that successfully reached a final state, one for table-as-matrix and another one for table-as-furniture.*

Search arises both in parsing and production. In parsing it arises because most word forms or syntactic constraints have multiple meanings and functions, and it is often not possible to make a definite choice until more of the sentence has been processed. Sometimes it is even necessary to work out multiple interpretations that will be disambiguated by the context. In production, search arises because there is usually more than one way to express a particular meaning, and it may not yet be possible to decide fully on a particular choice until other aspects of the sentence are worked out. This ambiguity is also why we see false starts, hesitations, and self-corrections in normal language production.

The search space is potentially explosive. Most words in a language have at least half a dozen meanings, and the form constraints of many grammatical constructions

are often shared with several other constructions. It is therefore computationally inefficient to exhaustively explore a search, and some sort of heuristic search method must be employed. By default, FCG uses a best-first search method based on scoring each node in the search space. (See the contribution by Bleys et al., 2011, later in this book). The score is based on the score of the constructions used so far in the chain, which is in turn based on their success in previous interactions, and on the degree with which each construction matches with the transient structure built so far.

The design of lexicons and grammars must take great care of avoiding search as much as possible. One of the main functions of syntactic and semantic categorizations is precisely to aid language users in avoiding search, which implies that as many constraints as possible must be included on the syntactic or semantic pole of constructions so that the best decision can be made as on whether to try out or proceed with a construction. This process usually involves thinking hard about the conditions of applicability of a construction and in particular about the interactions between slightly similar but competing constructions. It also requires thinking about how two constructions are cooperating to achieve a global purpose.

Adding `footprints` to a transient structure is another technique for avoiding search or the recursive applications of constructions, and it is also a primitive that can be used for many other issues such as the handling of defaults (see Beuls (2011)). When a construction applies, it can leave a kind of marker, called a *footprint*, and the application of the construction the second time around gets blocked, because the construction first checks whether its footprint is already there. Footprints are represented as unit features, attached to the unit concerned.

The problem of infinite application actually occurs with the mouse-construction shown earlier. This construction builds a new unit that contains the meaning and the form that was earlier in the top-unit, while respectively producing or parsing. It can again apply to this newly created unit, and then again to the unit that would be created from that, and so on. Footprints easily solve this problem, as illustrated with the following final form of the mouse-construction.

```
(def-cxn mouse-cxn ()
  ((?top-unit
     (tag ?meaning
          (meaning (== (mouse ?indiv))))
     (footprints (==0 mouse-cxn)))
   ((J ?mouse-unit ?top-unit)
     ?meaning
```

```
    (args (?mice-set ?context))
    (sem-cat
     (==1 (is-animate +) (class object)
          (is-countable +)))
    (footprints (==1 mouse-cxn))))
<==>
((?top-unit
   (tag ?form
        (form (== (string ?mouse-unit "mouse"))))
   (footprints (==0 mouse-cxn)))
 ((J ?mouse-unit ?top-unit)
   ?form
   (syn-cat
     (==1 (lex-cat noun) (number singular)))
     (footprints (==1 mouse-cxn)))))
```

The J-unit adds the footprint `mouse-cxn` to the mouse-unit it is creating and if the construction applies (again) it first checks to make sure that this footprint is not there. The same happens both on the semantic and syntactic side.

5.2. Construction Sets and Networks

When dealing with large lexicons and grammars, it is necessary to speed up the retrieval of those constructions that are potentially relevant, otherwise the FCG-interpreter would spend all its time just finding constructions that might apply. This necessary acceleration is done by organizing constructions into different *construction sets*, which apply as a group before another set is considered, and by introducing *networks* among constructions so that the successful application of one construction can prime others that are known from past processing to be potentially relevant. These networks can be built up automatically based on the actual usage of constructions. See Wellens (2011) for a further discussion of these various mechanisms and how they optimize construction application and influence the search process.

6. Templates

It is important to know what constructions look like and how they are processed in parsing and production. But grammar design will usually not be done at this level, simply because it would be too complicated and error-prone. Instead, the grammar

designer (and learning algorithms) use *templates* that capture *design patterns* that are needed for the language being studied. Similar approaches are common in other formalisms which use macros for writing recurrent grammatical patterns (Meurers, 2001). The set of possible templates is open-ended but a set of common default templates is provided with FCG implementations. This section briefly discusses what templates look like and how they are used to build constructions. All other papers in this book use templates and so many more concrete examples will be given.

A *template* has a number of *slots* which can either be unit-features or items that are translated into aspects of unit-features. The slots consist of symbols preceded by a semicolon. The general form of a template is as follows:

```
( template-name   construction-name
  :slot   value
  ...
  :slot   value)
```

The *construction-name* refers to the construction on which the template operates. To make it clear that we are dealing with a template, definitions are always drawn within a box.

Usually there is a template that defines the skeletal outline of a construction and then other templates build further on this skeleton, adding more features and possible more units to the semantic or syntactic pole. A new construction can also start out as a copy of an existing (more abstract) construction to which more elements are added that further constrain or embellish the construction, thus implementing inheritance between constructions.

A first example how a skeletal template is called provides the beginnings of a definition of the lexical construction for “mouse” that was discussed earlier:

```
(def-lex-skeleton mouse-cxn
  :meaning (== (mouse ?mice-set ?context))
  :args (?mice-set ?context)
  :string "mouse")
```

The `def-lex-skeleton` template only requires the grammar designer to specify the meaning, the word string, and which arguments in the meaning will be available to link the meaning supplied by the unit to meanings supplied by other units.

The def-lex-skeleton template expands the information supplied with its slots into the following operational definition of the construction. (Elements supplied by the template are in bold):

```
(def-cxn mouse-cxn ()
  ((?top-unit
     (tag ?meaning
          (meaning (== (mouse ?mice-set ?context))))
     (footprints (==0 mouse-cxn)))
   ((J ?mouse-unit ?top-unit)
     ?meaning
     (args (?mice-set ?context))
     (footprints (==1 mouse-cxn))))
  <==>
  ((?top-unit
     (tag ?form
          (form (== (string ?mouse-unit "mouse"))))
     (footprints (==0 mouse-cxn)))
   ((J ?mouse-unit ?top-unit)
     ?form
     (footprints (==1 mouse-cxn)))))
```

All the complexities in creating new units and tagging or moving parts of meaning and form are hidden in the template, and the grammar designer only has to consider the essentials, namely what is the meaning, what are the arguments and what is the string covered by this lexical construction.

Adding some syntactic and semantic categorizations to this lexical construction is done with another template called def-lex-cat. It specifies what semantic and syntactic categories are to be added:

```
(def-lex-cat mouse-cxn
  :sem-cat (==1 (is-animate +)
                (is-countable +)
                (class object))
  :syn-cat (==1 (lex-cat noun)
                (number singular)))
```

The def-lex-cat template is smart enough to work this information into the skeletal definition of the mouse construction created earlier. (The parts added by the template are shown in bold.)

```
(def-cxn mouse-cxn ()
  ((?top-unit
     (tag ?meaning
          (meaning (== (mouse ?mice-set ?context))))
     (footprints (==0 mouse-cxn)))
   ((J ?mouse-unit ?top-unit)
     ?meaning
     (args (?mice-set ?context))
     (sem-cat
      (==1 (is-animate +) (class object)
           (is-countable +)))
     (footprints (==1 mouse-cxn))))
  <==>
  ((?top-unit
     (tag ?form
          (form (== (string ?mouse-unit "mouse"))))
     (footprints (==0 mouse-cxn)))
   ((J ?mouse-unit ?top-unit)
     ?form
     (syn-cat
        (==1 (lex-cat noun) (number singular)))
     (footprints (==1 mouse-cxn)))))
```

All the templates that are concerned with the same construction are typically grouped together in an overarching template. For lexical constructions, this template is called `def-lex-cxn`. An example of the definition of a construction called `table-as-furniture-cxn` for the lexical item “table” is:

```
(def-lex-cxn table-as-furniture-cxn
    (def-lex-skeleton table-as-furniture-cxn
      :meaning
         (== (piece-of-furniture ?table-set ?context)
             (flat-surface ?table-set))
      :args (?table-set ?context)
      :string "table")
    (def-lex-cat table-as-furniture-cxn
      :sem-cat (==1 (is-animate -)
                    (is-countable +)
                    (class object))
      :syn-cat (==1 (lex-cat noun)
                    (number singular))))
```

Clearly this style of defining constructions brings more clarity and is closer to the more declarative way in which linguists like to study and define constructions. Later chapters in this book introduce a variety of templates that are now commonly used in FCG implementations for phrasal constructions, argument structure constructions, etc. The development of templates is a very active domain of research and there is no claim that the templates that will be used later form the definitive set, and neither that all languages share all templates.

7. Conclusions

This chapter contained some of the basic representational and processing mechanisms available in FCG. These mechanisms build further on proposals that have existed in the computational linguistics literature for decades but use them in novel ways. Learning FCG comes only from intense practice in using these various representational mechanisms and understanding their full impact on language processing. Often a simple solution only becomes apparent after working out multiple variations for the same problem. Carefully examining the case studies already carried out by others and looking at the design patterns captured in templates is a good way to learn, but mastering FCG requires doing many exercises oneself.

FCG has made a number of different design decisions as compared with other formalisms for construction grammar. The insistence on bi-directionality, the use of logic variables for structure sharing, footprints, and the building and expansion of hierarchical structures with the J-operator are some of the most important characteristic features of FCG. Whether these mechanisms are sufficient to deal with all the

remarkable phenomena found in human languages is too early to tell. Many more case studies need to be carried out to confront the formalism with the rich phenomena found in human natural languages. Whether these mechanisms provide the best solution is also too early to tell. At this stage in the development of (computational) construction grammar we should explore many avenues and work out many more concrete case studies to discover the fundamental linguistic representations and operations that could adequately explain how language is processed and learned. At the same time, the experiments in dialogue, language learning and language evolution that have already been carried out using FCG attest to the great power of the formalism and its versatility. They show that the construction grammar perspective need not be restricted to verbal descriptions of language phenomena only but can compete with other linguistic frameworks in terms of rigor and computational adequacy.

Acknowledgements

The research reported here was conducted at the Sony Computer Science Laboratory in Paris and the Artificial Intelligence Laboratory of the Free University of Brussels (VUB).

References

Baker, Collin, Charles Fillmore, John Lowe (1998). The berkeley framenet project. In *Proceedings of the COLING-ACL*. Montreal, Canada.

Beuls, Katrien (2011). Construction sets and unmarked forms: A case study for Hungarian verbal agreement. In Luc Steels (Ed.), *Design Patterns in Fluid Construction Grammar*. Amsterdam: John Benjamins.

Bleys, Joris, Kevin Stadler, Joachim De Beule (2011). Search in linguistic processing. In Luc Steels (Ed.), *Design Patterns in Fluid Construction Grammar*. Amsterdam: John Benjamins.

Carpenter, Bob (2002). *The logic of typed feature structures with applications to unification grammars, logic programs and constraint resolution*. Cambridge University Press.

Copestake, Ann, Dan Flickinger, Carl Pollard, Ivan Sag (2006). Minimal recursion semantics: an introduction. *Research on Language and Computation*, 3(4), 281–332.

Fanselow, Gisbert (2001). Features, theta-roles, and free constituent order. *Linguistic Inquiry*, 32(3), 405–437.

Gazdar, Gerald, Ewan Klein, Geoffrey Pullum, Ivan Sag (1985). *Generalized Phrase Structure Grammar.* Harvard University Press.

Haspelmath, Martin (2007). Pre-established categories don't exist: Consequences for language description and typology. *Linguistic Typology*, 11(1), 119–132.

Kay, Martin (1986). Parsing in functional unification grammar. In B.J. Grosz, K. Sparck-Jones, B. Webber (Eds.), *Readings in Natural Language Processing*. Morgan Kaufmann.

Meurers, Detmar (2001). On expressing lexical generalizations in hpsg. *Nordic Journal of Linguistics*, 24(2), 161–217.

Norvig, Peter (1992). *Paradigms of Artificial Intelligence Programming. Case Studies in Common Lisp.* San Francisco: Morgan Kauffman.

Steels, Luc (2011). A design pattern for phrasal constructions. In Luc Steels (Ed.), *Design Patterns in Fluid Construction Grammar*. Amsterdam: John Benjamins.

Steels, Luc (Ed.) (2012). *Experiments in Cultural Language Evolution*. Amsterdam: John Benjamins.

Steels, Luc, Joachim De Beule (2006). Unify and merge in fluid construction grammar. In P. Vogt, Y. Sugita, E. Tuci, C. Nehaniv (Eds.), *Symbol Grounding and Beyond: Proceedings of the Third International Workshop on the Emergence and Evolution of Linguistic Commun*, LNAI 4211, 197–223. Berlin: Springer-Verlag.

van Trijp, Remi (2011). Feature matrices and agreement: A case study for German case. In Luc Steels (Ed.), *Design Patterns in Fluid Construction Grammar*. Amsterdam: John Benjamins.

Wellens, Pieter (2011). Organizing constructions in networks. In Luc Steels (Ed.), *Design Patterns in Fluid Construction Grammar*. Amsterdam: John Benjamins.