# Codeco: A Practical Notation for Controlled English Grammars in Predictive Editors*

Tobias Kuhn[1,2]

[1] Department of Informatics & Institute of Computational Linguistics,
University of Zurich, Switzerland
[2] Department of Intelligent Computer Systems,
University of Malta
kuhntobias@gmail.com
http://www.tkuhn.ch

**Abstract.** This paper introduces a new grammar notation, called Codeco, designed for controlled natural language (CNL) and predictive editors. Existing grammar frameworks that target either formal or natural languages do not work out particularly well for CNL, especially if they are to be used in predictive editors and if anaphoric references should be resolved in a deterministic way. It is not trivial to build predictive editors that can precisely determine which anaphoric references are possible at a certain position. This paper shows how such complex structures can be represented in Codeco, a novel grammar notation for CNL. Two different parsers have been implemented (one in Prolog and another one in Java) and a large subset of Attempto Controlled English (ACE) has been represented in Codeco. The results show that Codeco is practical, adequate and efficient.

## 1 Introduction

Controlled natural languages (CNL) have been proposed to make knowledge representation systems more user-friendly [25,15]. The scope of this work is restricted to controlled natural languages with a direct and deterministic mapping to some kind of formal logic, like Attempto Controlled English (ACE) [8].

One of the main problems of CNL is that it can be very difficult for users to write statements that comply with the restrictions of CNL. Three approaches have been proposed so far to solve this problem: error messages [3], language generation [24], and predictive editors [26,15]. Each of the approaches has its own advantages and drawbacks. The work presented here follows the predictive editor approach, where the editor shows all possible continuations of a partial sentence. However, implementing lookahead features (i.e. retrieving the possible continuations for a given partial sentence on the basis of a given grammar) is not

* The work presented here was funded by the research grant (Forschungskredit) programs 2006 and 2008 of the University of Zurich.

M. Rosner and N.E. Fuchs (Eds.): CNL 2010, LNAI 7175, pp. 95–114, 2012.
© Springer-Verlag Berlin Heidelberg 2012

a trivial task, especially if the grammar describes complex nonlocal structures like anaphoric references. Furthermore, good tool integration is crucial for CNL, no matter which approach is followed. For this reason, it is desirable to have a simple grammar notation that is fully declarative and can easily be implemented in different kinds of programming languages.

In the remainder of this paper, the relevant background is discussed and the problems we are facing are explained (Sect. 2). Then, the Codeco grammar notation is introduced, which is designed to solve the identified problems (Sect. 3). Next, two implementations of Codeco are presented (Sect. 4) and a specific grammar written in this notation is introduced (Sect. 5). Finally, several evaluation results of the approach are shown (Sect. 6) and the conclusions are drawn (Sect. 7).

This paper is built from a chapter of the author's doctoral thesis [17], which gives more information about the details of the approach and about the concrete application in specific tools. The Codeco notation was outlined in the extended abstract of this full paper [16].

## 2 Background

Below, a number of requirements for controlled English grammars are introduced. Then, different kinds of existing grammar frameworks are discussed: frameworks for natural languages, parser generators, and definite clause grammars (DCG). Finally, related work is introduced in the form of the Grammatical Framework.

### 2.1 Requirements for Controlled English Grammars

What follows is a list of requirements that grammars of controlled English have to meet if they are to be defined, implemented, used, and reused efficiently, under the assumption that the predictive editor approach is followed and that the language supports complex phenomena like anaphoric references.

**Concreteness.** Concreteness is an obvious requirement. Due to their practical and computer-oriented nature, CNL grammars should be concrete, i.e. fully formalized to be read and interpreted by computer programs.

**Declarativeness.** As a second requirement, CNL grammars should be declarative, i.e. defined in a way that does not depend on a concrete algorithm or implementation. Declarative grammars can be completely separated from the parser that processes them. This makes it easy for such grammars to be used by other programs, to replace the parser, or to have different parsers for the same language. Declarative grammars are easy to change and reuse and can be shared easily between different parties using the same CNL.

**Lookahead Features.** Predictive editors require the availability of lookahead features, i.e. the possibility to find out how a partial text can be continued. For this reason, CNLs must be defined in a form that enables the efficient implementation of such lookahead features. Concretely, this means that a partial text, for instance "a brother of Sue likes ...", can be given to the parser and that the parser is able return the complete set of words that can be used to continue the partial sentence according to the grammar. For the given example, the parser might say that "a", "every", "no", "somebody", "John", "Sue", "himself" and "her" are the possibilities to continue the partial sentence.

**Anaphoric References and Scopes.** CNLs that support anaphoric references raise special requirements. For such languages, it should be possible to describe the circumstances under which anaphoric references are allowed in an exact, declarative, and simple way that — in order to have a clear separation of syntax and semantics — does not depend on the semantic representation. Concretely, a CNL should allow the use of a referential expression like "it" only if a matching antecedent (e.g. "a country") can be identified in the preceding text. Every sentence that contains an expression that can only be interpreted in a referential way but cannot be resolved must be considered syntactically incorrect. In other words, CNL grammars must be able to represent the fact that only resolvable anaphoric references are allowed.

The resolvability of anaphoric references depends on the scopes of the preceding text. Scopes are raised by certain structures like negation, and they cover certain areas of the text that denote the range of influence of the respective expression. While scopes in natural language can be considered a semantic phenomenon, they have to be treated as a syntactic issue in CNLs if the restrictions on anaphoric references are to be described appropriately. Thus, a grammar that defines the syntax of a CNL needs to specify anaphoric references, their antecedents, and the positions at which scopes are opened and closed.

**Implementability.** Finally, a CNL grammar notation should be easy to implement in different programming languages. As a consequence, a CNL grammar notation should be neutral with respect to the programming paradigm of its parser.

The implementability requirement is motivated by the fact that the usability of CNL heavily depends on good integration into user interfaces like predictive editors. For this reason, it is desirable that the CNL parser is implemented in the same programming language as the user interface component.

Another reason why implementability is important is that the parser is often not the only tool that needs to know the CNL grammar. There can be many other tools besides the parser that need to read and process the grammar, e.g. editors [26], paraphrasers [12] and verbalizers[1]. Furthermore, more than one parser might be necessary for practical reasons. For example, a simple top-down parser may be the best option for simple grammars when used for parsing large texts

[1] see e.g. `http://attempto.ifi.uzh.ch/site/docs/owl_to_ace.html`

in batch mode and for doing regression tests (e.g. through language generation). On the other hand, a chart parser is better suited for complex grammars and for providing lookahead capabilities.

### 2.2 Natural Language Grammar Frameworks

A large number of grammar frameworks exist to process natural languages. Some of the most popular ones are *Head-Driven Phrase Structure Grammars* (HPSG) [23], *Lexical-Functional Grammars* [13], and *Tree-Adjoining Grammars* [11]. More of them are discussed by Cole et al. [5]. Most of these frameworks are defined in an abstract and declarative way. Concrete grammar definitions based on such frameworks, however, are often not fully declarative.

Despite many similarities, a number of important differences between natural language grammars and grammars for CNLs can be identified that have the consequence that the grammar frameworks for natural languages do not work out very well for CNLs. Most of the differences originate from the fact that the two kinds of grammars are the results of opposing goals. Natural language grammars are *language descriptions*: they describe existing phenomena. CNL grammars, in contrast, are *language definitions*: they define something new.

Obviously, grammars for natural languages and those for CNLs differ in complexity. Natural languages are very complex and so must be the grammars that thoroughly describe such languages. CNLs are typically much simpler and abandon natural structures that are difficult to process.

Partly because of the high degree of complexity, providing lookahead features on the basis of those frameworks is very hard. Another reason is that lookahead features are simply not relevant for natural language applications, and thus no special attention has been given to this problem. The difficulty of implementing lookahead features with natural language grammar frameworks can be seen by the fact that no predictive editors exist for CNLs that emerged from an NLP background like CPL [4] or CLOnE [9].

The handling of ambiguity is another important difference. Natural language grammars have to deal with the inherent ambiguity of natural languages. Context information and background knowledge can help resolving ambiguities, but there is always a remaining degree of uncertainty. Natural language grammar frameworks are designed to be able to cope with such situations, can represent structural ambiguity by using underspecified representations, and require the parser to disambiguate by applying heuristic methods. In contrast, CNLs (the formal ones on which this paper focuses) remove ambiguity by their design, which typically makes underspecification and heuristics unnecessary.

Finally, the resolution of anaphoric references to appropriate antecedents is another particularly difficult problem for the correct representation of natural language. In computational linguistics, this problem is usually solved by applying complex algorithms to find the most likely antecedents (see e.g. [19]). The following example should clarify why this is such a difficult problem: An anaphoric pronoun like "it" can refer to a noun phrase that has been introduced in the preceding text but it can also refer to a broader structure like a complete

sentence or paragraph. It is also possible that "it" refers to something that has been introduced only in an implicit way or to something that will be identified only in the text that follows later. Furthermore, "it" can refer to something outside of the text, meaning that background knowledge is needed to resolve it. Altogether, this has the consequence that sentences like "an object contains it" have to be considered syntactically correct even if no matching antecedent for "it" can be clearly identified in the text. In order to address the problem of anaphoric references, natural language grammar frameworks like HPSG establish "binding theories" [2,23] that consist of principles that describe under which circumstances two components of the text can refer to the same thing. Applying these binding theories, however, just gives a set of possible antecedents for each anaphor and does not deal with deterministic resolution of them.

### 2.3 Parser Generators

A number of systems exist that are aimed at the definition and parsing of formal languages (e.g. programming languages). In the simplest case, such grammars are written in Backus-Naur Form [21,14]. Examples of more sophisticated grammar formalisms for formal languages — called *parser generators* — include Yacc [10] and GNU bison[2]. Some CNLs like Formalized-English [20] are defined in such parser generator notations.

The general problem of these formalisms is that context-sensitive constraints cannot be defined in a declarative way. Simple context-free languages can be described in a declarative and simple way by using plain Backus-Naur style grammars. However, such grammars are very limited and even very simple CNLs cannot be defined appropriately. It is possible to describe more complex grammars containing context-sensitive elements with such parser generators. However, this has to be done in the form of procedural extensions that depend on a particular programming language to be interpreted. Thus, the property of declarativeness gets lost when more complex languages are described.

When discussing lookahead capabilities of parser generators, it has to be noted that the term *lookahead* has a different meaning in the context of parser generators: *lookahead* denotes how far the parsing algorithm looks ahead in the fixed token list before deciding which rule to apply. Lookahead in our sense of the word — i.e. predicting possible next tokens — is not directly supported by existing parser generators. However, as long as no procedural extensions are used, this is not hard to implement. Actually, a simple kind of lookahead (in our sense of the word) is available in many source code editors in the form of code completion features.

### 2.4 Definite Clause Grammars

Definite clause grammars (DCG) [22], finally, are a simple but powerful way to define grammars for natural and formal languages and are mostly written in

[2] `http://www.gnu.org/software/bison/`

logic-based programming languages like Prolog. In fact, many of the grammar frameworks for natural languages introduced above are usually implemented on the basis of Prolog DCGs.

In their core, DCGs are fully declarative and can thus in principle be processed by any programming language. Since they build upon the logical concept of definite clauses, they are easy to process for logic-based programming languages. In other programming languages, however, a large overhead is necessary to simulate backtracking and unification.

DCGs are good in terms of expressivity because they are not necessarily context-free but can contain context-sensitive elements. Anaphoric references, however, are again a problem. Defining them in an appropriate way is difficult in plain DCG grammars. The following two exemplary grammar rules show how antecedents and anaphors could be defined:

```
np(Agr, Ante-[Agr|Ante]) --> determiner(Agr), noun(Agr).
np(Agr, Ante-Ante) --> ana_pron(Agr), { once(member(Agr,Ante)) }.
```

The code inside the curly brackets defines that the agreement structure of the pronoun is unified with the first possible element of the antecedent list.

This approach has some problems. First of all, the curly brackets contain code that is not fully declarative. A more serious problem, however, is the way how connections between anaphors and antecedents are established. The accessible antecedents are passed through the grammar by using input and output lists of the form "`In-Out`" so that new elements can be added to the list whenever an antecedent occurs in the text. The problem that follows from this approach is that the definition of anaphoric references cannot be done locally in the grammar rules that actually deal with anaphoric structures but they affect almost the complete grammar, as illustrated by the following example:

```
s(Ante1-Ante3) --> np(Agr, Ante1-Ante2), vp(Agr, Ante2-Ante3).
```

As this example shows, anaphoric references also have to be considered when writing grammar rules that have otherwise nothing to do with anaphors or antecedents. This is neither convenient nor elegant.

Different DCG extensions have been proposed in order to describe natural language in a more appropriate way. Assumption Grammars [6], for example, are motivated by natural language phenomena that are hard to express otherwise, like free word order. Simple anaphoric references can be represented in a very clean way, but there are problems with more complex anaphor types like irreflexive pronouns.

A further problem with the DCG approach concerns lookahead features. In principle, it is possible to provide lookahead features with standard Prolog DCGs, as shown in previous work conducted with Rolf Schwitter [18]. However, this approach is not very efficient and can become impractical for complex grammars and long sentences.

### 2.5 Related Work

The Grammatical Framework (GF) [1] is the only existing grammar framework (to the author's knowledge) that has a specific focus on CNL. Apart from the fact that it has no particular support for describing the resolvability of anaphoric references on the syntactic level, GF fulfills the requirements introduced above. With extensions (perhaps inspired by Codeco), it could become a suitable grammar notation for the specific problem described in this paper.

## 3 The Codeco Notation

On the basis of the aforementioned requirements, a grammar notation called *Codeco*, which stands for "*co*ncrete and *de*clarative grammar notation for *co*ntrolled natural languages", is introduced here.

The Codeco notation has been developed with ACE in mind, and the elements of Codeco will be motivated by ACE examples. Nevertheless, this notation should be general enough for other controlled subsets of English, and for controlled subsets of other languages. Codeco can be conceived as a proposal for a general CNL grammar notation. As I will show, it works well for a large subset of ACE, but it cannot be excluded that extensions or modifications would become necessary to be able to express the syntax of other CNLs.

Below, the different elements of the Codeco notation are introduced, i.e. grammar rules, grammatical categories, and certain special elements. After that, the issue of reference resolution is discussed.

### 3.1 Simple Categories and Grammar Rules

Grammar rules in Codeco use the operator "$\overset{:}{\rightarrow}$" (where the colon on the arrow is needed to distinguish normal rules from scope-closing rules as they will be introduced later on):

$$vp \quad \overset{:}{\longrightarrow} \quad v \quad np$$

Terminal categories are represented in square brackets:

$$v \quad \overset{:}{\longrightarrow} \quad [\,\text{does not}\,] \quad verb$$

In order to provide a clean interface between grammar and lexicon, Codeco has a special notation for pre-terminal categories, being marked with an underline:

$$np \quad \overset{:}{\longrightarrow} \quad [\,\text{a}\,] \quad \underline{noun}$$

Pre-terminal categories are conceptually somewhere between non-terminal and terminal categories, in the sense that they can be expanded but only to terminal categories. This means that pre-terminal categories can occur on the left hand side of a rule only if the right hand side consists of exactly one terminal category.

Such rules are called *lexical rules* and are represented with a plain arrow, for instance:

$$\underline{noun} \;\rightarrow\; [\,\text{person}\,]$$

Lexical rules can be stored in a dynamic lexicon but they can also be part of the static grammar.

In order to support context-sensitivity, non-terminal and pre-terminal categories can be augmented with flat feature structures. They are shown here using the colon operator ":" with the name of the feature to the left and its value to the right. Values can be variables, which are displayed as boxes:

$$vp\left(\begin{matrix}\text{num: }\boxed{\text{Num}}\\ \text{neg: }\boxed{\text{Neg}}\end{matrix}\right) \;\overset{\vdots}{\longrightarrow}\; v\left(\begin{matrix}\text{num: }\boxed{\text{Num}}\\ \text{neg: }\boxed{\text{Neg}}\\ \text{type: tr}\end{matrix}\right)\; np\Big(\text{case: acc}\Big)$$

$$v\left(\begin{matrix}\text{neg: }+\\ \text{type: }\boxed{\text{Type}}\end{matrix}\right) \;\overset{\vdots}{\longrightarrow}\; [\,\text{does not}\,]\;\; verb\Big(\text{type: }\boxed{\text{Type}}\Big)$$

$$np\Big(\text{noun: }\boxed{\text{Noun}}\Big) \;\overset{\vdots}{\longrightarrow}\; [\,\text{a}\,]\;\; \underline{noun}\Big(\text{text: }\boxed{\text{Noun}}\Big)$$

An important restriction is that feature values *cannot* be feature structures themselves. This means that feature structures in Codeco are always flat. This restriction has practical reasons. It should keep Codeco simple and easy to implement, but it can easily be dropped in theory.

### 3.2 Normal Forward and Backward References

So far, the introduced elements of Codeco are quite straightforward and not very specific to CNL or predictive editors. The support for anaphoric references, however, requires some novel extensions.

In principle, it is easy to support sentences like

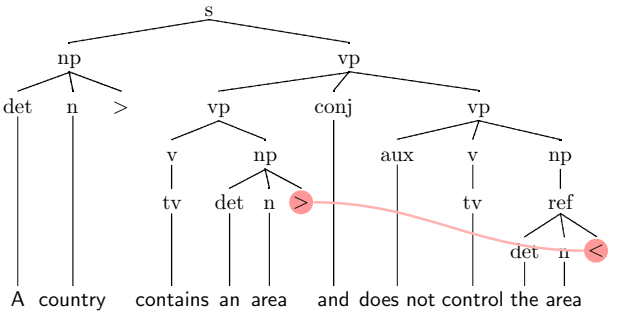
A country contains an area that is not controlled by the country.

where "the country" is a resolvable anaphoric reference. However, given that we only have the Codeco elements introduced so far, it is not possible to suppress sentences like

Every area is controlled by the country.

where "the country" is not resolvable. This can be acceptable, but in many situations there are good reasons to disallow such non-resolvable references.

In Codeco, the use of anaphoric references can be restricted to positions where they can be resolved. This is done with the help of the special categories ">" and "<", which describe nonlocal dependencies across the syntax tree, as the following illustration shows:




“>” represents a *forward reference* and marks a position in the text to which anaphoric references can refer, i.e. “>” stands for antecedents. “<” represents a *backward reference* and refers back to the closest possible antecedent, i.e. “<” stands for anaphors. These special categories can have feature structures and they can occur only in the body of rules, for example:

$$np \overset{\vdots}{\longrightarrow} [\,\text{a}\,] \quad \underline{noun}\left(\text{text: } \boxed{\text{Noun}}\right) \; > \begin{pmatrix} \text{type: noun} \\ \text{noun: } \boxed{\text{Noun}} \end{pmatrix}$$

$$ref \overset{\vdots}{\longrightarrow} [\,\text{the}\,] \quad \underline{noun}\left(\text{text: } \boxed{\text{Noun}}\right) \; < \begin{pmatrix} \text{type: noun} \\ \text{noun: } \boxed{\text{Noun}} \end{pmatrix}$$

The forward reference of the first rule establishes an antecedent to which later backward references can refer. The second rule contains such a backward reference that refers back to an antecedent with a matching feature structure. In this example, forward and backward references have to agree in their type and their noun (represented by the features “type” and “noun”). This has the effect that “the country”, for example, can refer to “a country”, but “the area” cannot.

Forward references always succeed, whereas backward references succeed only if a matching antecedent in the form of a forward reference can be found somewhere to the left in the syntax tree.

In order to distinguish these simple types of forward and backward references from other reference types that will be introduced below, they are called *normal forward references* and *normal backward references*, respectively.

Altogether, these special categories provide a very simple way to establish nonlocal dependencies in the grammar for describing anaphoric references. However, as we will discover, these simple kinds of references are not general enough for all types of references we would like to represent. We need more reference types, but first accessibility constraints have to be discussed.

### 3.3 Scopes and Accessibility

As already pointed out, anaphoric references are affected by scopes. References are resolvable only to positions in the previous text that are accessible, i.e. that are not inside closed scopes. An example is

Every man protects a house from every enemy and does not destroy ...

where one can refer to "man" and to "house" but not to "enemy" (because "every" opens a scope that is closed after "enemy"). The Codeco elements introduced so far do not allow for such restrictions. Additional elements are needed to define where scopes open and where they close.

The position where a scope opens is represented in Codeco by the special category "//" called *scope opener*, for example:

$$quant\Big(\text{exist: }-\Big) \xrightarrow{\vdots} \; // \quad [\,\text{every}\,]$$

Scopes that are open have to be closed somewhere. In contrast to the opening positions of scopes, their closing positions can be far away from the scope-triggering structure. For this reason, the closing positions of scopes cannot be defined in the same way as their opening positions. Instead, the positions where scopes close are defined in Codeco by the use of scope-closing rules "$\xrightarrow{\sim}$", for instance:

$$vp\Big(\text{num: }\boxed{\text{Num}}\Big) \xrightarrow{\sim} v\begin{pmatrix}\text{neg: }+\\ \text{num: }\boxed{\text{Num}}\\ \text{type: tr}\end{pmatrix} np\Big(\text{case: acc}\Big)$$

This rule states that any scope that is opened by the direct or indirect children of "*v*" and "*np*" is closed at the end of "*np*". If no scopes have been opened, scope-closing rules simply behave like normal rules.

In contrast to most other approaches, scopes are defined in a way that is completely independent from the semantic representation.

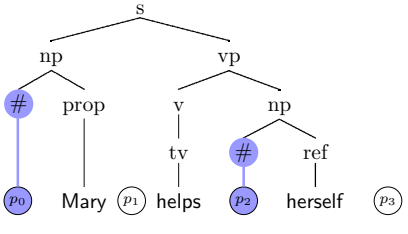
### 3.4 Position Operators

With the introduced Codeco elements, anaphoric definite noun phrases like "the area" can be restricted to positions where they are resolvable. However, it is not possible so far to define, for example, that a reflexive pronoun like "herself" is allowed only if it refers to the subject of the respective verb phrase. Concretely, we cannot distinguish the following two cases:

A woman helps herself.
* A woman knows a man who helps herself.

The problem is that there is no way to check whether a potential antecedent is the subject of a given anaphoric reference or not. What is needed is a way of assigning an identifier to each antecedent.

To this aim, Codeco employs the position operator "#", which takes a variable and assigns it an identifier that represents the respective position in the text. The following picture visualizes how position operators work:



With the use of position operators, reflexive pronouns can be defined in a way that excludes unresolvable pronouns, i.e. excludes pronouns that do not match with the subject of the given verb phrase:

$$np\left(\text{id: } \boxed{\text{Id}}\right) \overset{:}{\longrightarrow} \#\boxed{\text{Id}} \quad prop\left(\text{human: } \boxed{\text{H}}\right) > \begin{pmatrix}\text{id: } \boxed{\text{Id}}\\ \text{human: } \boxed{\text{H}}\\ \text{type: prop}\end{pmatrix}$$

$$ref\left(\text{subj: } \boxed{\text{Subj}}\right) \overset{:}{\longrightarrow} [\,\text{itself}\,] < \begin{pmatrix}\text{id: } \boxed{\text{Subj}}\\ \text{human: } -\end{pmatrix}$$

Position operators allow us to use identifiers — e.g. for identifying the subject of a verb phrase — in a very simple and declarative way. As we will see, however, a further extension is needed for the appropriate definition of irreflexive pronouns.

### 3.5 Negative Backward References

A further problem that has to be solved concerns variables as they are supported, for instance, by ACE. Phrases like "a person X" can be used to introduce a variable "X". A problem arises if the same variable is introduced twice:

* A person X knows a person X.

One solution is to allow such sentences and to define that the second introduction of "X" overrides the first one so that subsequent occurrences of "X" can only refer to the second one. In first-order logic, for example, variables are treated this way. In CNL, however, the overriding of variables can be confusing to the readers. ACE, for example, does not allow variables to be overridden.

Such restrictions cannot be defined with the Codeco elements introduced so far. Another extension is needed: the special category "$\not<$" that can be used to ensure that there is no matching antecedent. This special category establishes *negative backward references*, which can be used — among other things — to ensure that no variable is introduced twice:

$$newvar \overset{:}{\longrightarrow} \underline{var}\left(\text{text: } \boxed{\text{V}}\right) \not< \begin{pmatrix}\text{type: var}\\ \text{var: } \boxed{\text{V}}\end{pmatrix} > \begin{pmatrix}\text{type: var}\\ \text{var: } \boxed{\text{V}}\end{pmatrix}$$

The special category "$\not<$" succeeds only if there is no accessible forward reference that unifies with the given feature structure.

### 3.6 Complex Backward References

The introduced Codeco elements are still not sufficient for expressing all the things we would like to express. As already mentioned, there is still a problem with irreflexive pronouns like "him". While reflexive pronouns like "himself" can be restricted to refer only to the respective subject, the introduced Codeco elements do not allow for preventing irreflexive pronouns from referring to the subject as well:

John knows Bill and helps him.
* John helps him.

These two cases cannot be distinguished so far. It thus becomes necessary to introduce *complex backward references*, which use the special structure "$<^{+}\ldots\ ^{-}\ldots$". Complex backward references can have several feature structures: one or more positive ones (after the symbol "$^{+}$"), which define how a matching antecedent must look like, and zero or more negative ones (after "$^{-}$"), which define how the antecedent must *not* look like. The symbol "$^{-}$" can be omitted if no negative feature structures are present.

In this way, irreflexive pronouns can be correctly represented, for instance:

$$ref\left(\text{subj: }\boxed{\text{Subj}}\right) \overset{\vdots}{\longrightarrow} [\,\text{he}\,] <^{+}\begin{pmatrix}\text{human: }+\\ \text{gender: masc}\end{pmatrix}^{-}\left(\text{id: }\boxed{\text{Subj}}\right)$$

Complex backward references refer to the closest accessible forward reference that unifies with one of the positive feature structures but is not unifiable with any of the negative ones.

Complex backward references are a powerful construct, with which anaphoric references can be restricted in a very general way. The following example — which is rather artificial and would probably not be very useful in practice — illustrates the general nature of complex backward references: One might want to define that the word "this" can be used to refer to something which is neuter and has no variable attached or which is a relation (whatever that means), but which is not the subject and is not a proper name. This complex behavior could be achieved with the following rule:

$$ref\left(\text{subj: }\boxed{\text{Subj}}\right) \overset{\vdots}{\longrightarrow} [\,\text{this}\,] <^{+}\begin{pmatrix}\text{hasvar: }-\\ \text{human: }-\end{pmatrix}\left(\text{type: relation}\right)^{-}\left(\text{id: }\boxed{\text{Subj}}\right)\left(\text{type: prop}\right)$$

Complex backward references that have exactly one positive feature structure and no negative ones are equivalent to normal backward references.

### 3.7 Strong Forward References

Finally, one last extension is needed in order to handle antecedents that are not affected by the accessibility constraints. For example, proper names are usually considered accessible even if under negation:

Mary does not love Bill. Mary hates him.

In such situations, the special category "$\gg$" can be used, which introduces a *strong forward reference*:

$$np\left(\text{id: }\boxed{\text{Id}}\right) \overset{\vdots}{\longrightarrow} prop\left(\text{human: }\boxed{\text{H}}\right) \gg \begin{pmatrix}\text{id: }\boxed{\text{Id}}\\ \text{human: }\boxed{\text{H}}\\ \text{type: prop}\end{pmatrix}$$

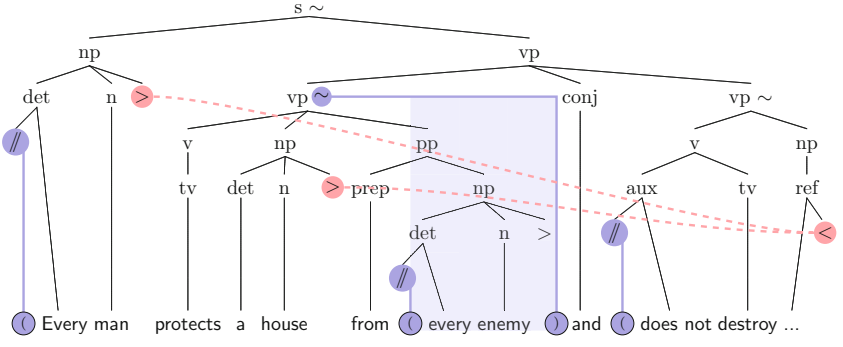
Strong forward references are always accessible even if they are within closed scopes. Apart from that, they behave like normal forward references.

### 3.8 Principles of Reference Resolution

The resolution of references in Codeco requires some more explanation. All three types of backward references (normal, negative and complex ones) are resolved according to the three principles of accessibility, proximity and left-dependence.

**Accessibility.** The principle of accessibility states that one can refer to forward references only if they are accessible from the position of the backward reference. A forward reference is accessible only if it is not within a scope that has been closed before the position of the backward reference, or if it is a strong forward reference.

This accessibility constraint can be visualized in the syntax tree. The syntax tree for the partial sentence shown in Section 3.3 could look as follows:

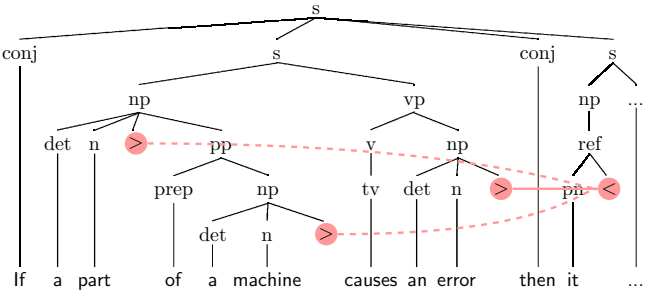



All nodes that represent the head of a scope-closing grammar rule are marked with "∼". The positions in the text where scopes open and close are marked by parentheses. In this example, three scopes have been opened but only the second one (the one in front of "every enemy") has been closed (after "enemy"). The shaded area marks the part of the syntax tree that is covered by this closed scope.

As a consequence of the accessibility constraint, the forward references for "man" and "house" are accessible from the position of the backward reference at the very end of the shown partial sentence. In contrast, the forward reference for "enemy" is not accessible because it is inside a closed scope. The possible references are shown as dashed lines. Thus, the partial sentence can be continued by the anaphoric references "the man" or "the house" (or equivalently "himself" or "it", respectively) but not by the reference "the enemy".

**Proximity.** Proximity is the second principle for the resolution of backward references. If a backward reference could potentially point to more than one forward reference then, as a last resort, the principle of proximity defines that the textually closest forward reference is taken. Or more precisely, when traversing the syntax tree starting from the backward reference and going back in a right-to-left, depth-first manner, the first matching and accessible forward reference is taken. This ensures that every backward reference resolves deterministically to exactly one forward reference.

In the following example, the reference "it" could in principle refer to three antecedents:




The pronoun "it" could refer to "part", "machine", or "error". According to the principle of proximity, the closest antecedent is taken, i.e. "error".

**Left-Dependence.** The principle of left-dependence, finally, means that everything to the left of a backward reference is considered for its resolution but everything to its right is not. The crucial point is that variable bindings entailed by a part of the syntax tree to the left of the reference are considered for its resolution, whereas variable bindings that would be entailed by a part of the syntax tree to the right are not considered.

The following example illustrates why the principle of left-dependence is important:

$$ref \xrightarrow{\vdots} [\,\text{the}\,] \; < \begin{pmatrix} \text{type: noun} \\ \text{noun: } \boxed{\text{N}} \end{pmatrix} \; \underline{noun}\left(\text{text: } \boxed{\text{N}}\right)$$

$$ref \xrightarrow{\vdots} [\,\text{the}\,] \; \underline{noun}\left(\text{text: } \boxed{\text{N}}\right) \; < \begin{pmatrix} \text{type: noun} \\ \text{noun: } \boxed{\text{N}} \end{pmatrix}$$

These are two versions of the same grammar rule. The only difference is that the backward reference and the pre-terminal category "*noun*" are switched. The first version is not a very sensible one: the backward reference is resolved without considering how the variable "N" is bound by the category "*noun*". The second version is much better: the resolution of the reference takes into account which noun has been read from the input text.

As a rule of thumb, backward references should generally follow the textual representation of the anaphoric reference and not precede it.

### 3.9 Restriction on Backward References

In order to provide proper and efficient lookahead algorithms that can handle backward references, their usage must be restricted: Backward references must immediately follow a terminal or pre-terminal category in the body of grammar rules. Thus, they are not allowed at the initial position of the rule body and they must not follow a non-terminal category.

However, the algorithms that process Codeco also work for grammars that do not follow this restriction. Only the lookahead feature would not work as expected.

## 4 Implementations

The Codeco notation currently has two implementations: one that transforms it into Prolog DCG grammars to be executed as Prolog programs, and another one that executes Codeco grammars in Java with a chart parsing algorithm.

The Prolog DCG implementation is primarily intended for running parsing and generation tasks in batch mode. As we will see, it is very fast, but it lacks lookahead features. Chart parsers are a good choice for such lookahead features, and this is the approach of the second implementation: Codeco grammars are executed in Java following the Earley chart parsing algorithm [7]. The Java implementation is slower than the one in Prolog (see below), but it has lookahead support (i.e. it can list all possible next tokens for partial sentences) and enables Java applications to run Codeco grammars without Prolog.

Both implementations can not only be used to parse but also to automatically generate all syntactically correct sentences up to a certain sentence length. The details of these implementations, including the lookahead algorithm and the necessary extensions for the Earley algorithm, can be found in the author's doctoral thesis [17].

The web applications AceWiki [15] and the ACE Editor[3] both use the Codeco notation. The Prolog implementation is used for regression testing, and the Java implementation is used by their predictive editors for the lookahead features.

## 5 ACE Codeco Grammar

The introduced Codeco notation has been used to describe a large subset of ACE, to be called *ACE Codeco*. This grammar consists of 164 grammar rules[4] and is used by the ACE Editor mentioned above.

The ACE Codeco grammar covers a large part of ACE including countable nouns, proper names, intransitive and transitive verbs, adjectives, adverbs, prepositions, plurals, negation, comparative and superlative adjectives and adverbs, *of*-phrases, relative clauses, modality, numerical quantifiers, coordination of sentences / verb phrases / relative clauses, conditional sentences, and questions. Anaphoric references are possible by using simple definite noun phrases, variables, and reflexive and irreflexive pronouns. However, there are some considerable restrictions with respect to the full language of ACE. Mass nouns, measurement nouns, ditransitive verbs, numbers and strings as noun phrases, sentences as verb phrase complements, Saxon genitive, possessive pronouns, noun phrase coordination, and commands are not covered at this point.

Nevertheless, this subset of ACE defined by the Codeco grammar is probably the broadest unambiguous subset of English that has ever been defined in a concrete and fully declarative way and that includes complex issues like anaphoric references.

[3] `http://attempto.ifi.uzh.ch/webapps/aceeditor/`

[4] See [17] for the complete grammar.

When sentences are generated for evaluation purposes from a grammar in an exhaustive manner then one quickly encounters a combinatorial explosion on the number of generated sentences. In practice, this means that one can only use a subset of the grammar for such evaluations. Such an evaluation subset has been defined for ACE Codeco, using a minimal lexicon and only 97 of the 164 grammar rules. These 97 grammar rules are chosen in a way that reduces combinatorial explosion but retains the complexity of the language.

## 6 Evaluation

On the basis of the ACE Codeco grammar and its evaluation subset, a number of tests can be performed. On the one hand, it can be evaluated whether the language described by ACE Codeco has the desired properties. On the other hand, it can be taken as a test case to evaluate the Codeco notation and the two implementations thereof.

**Ambiguity Check of ACE Codeco.** Languages like ACE are designed to be unambiguous on the syntactic level. This means that every valid sentence must have exactly one syntax tree according to the given grammar. By exhaustive language generation, the resulting sentences can be checked for duplicates. Sentences generated more than once have more than one possible syntax tree and are thus ambiguous.

Up to the length of ten tokens, the evaluation subset of ACE Codeco generates 2'250'869 sentences, which are all distinct. Thus, at least a large subset of ACE Codeco is unambiguous for at least relatively short sentences.

**Subset Check of ACE Codeco and Full ACE.** The ACE Codeco grammar is designed as a proper subset of ACE. It can now be checked automatically whether this is the case, at least for the evaluation subset of ACE Codeco and up to a certain sentence length.

Every sentence up to the length of ten tokens was submitted to the ACE parser (APE) and parsing succeeded in all cases. Since APE is the reference implementation of ACE, this means that these sentences are syntactically correct ACE sentences.

**Equivalence Check of the Implementations.** Since both implementations support language generation, we can check whether the two implementations accept the same set of sentences, as they should, for the ACE Codeco grammar.

The Java implementation has been used to generate all sentences up to the sentence length of eight tokens. Since the Java implementation is slower than the one based on Prolog DCGs (see below), the former cannot generate as long sentences as the latter within reasonable time. The resulting set of sentences generated by the Java implementation is identical to the one generated by the Prolog DCG. This is an indication that the two implementations contain no major bugs and that they interpret Codeco grammars in the same way.

**Table 1.** These are the results of a performance test of the two implementations of Codeco. As a comparison, the performance of the existing ACE parser (APE) is shown for the parsing task.

| task | grammar | implementation | time in seconds *overall* | *average* |
|---|---|---|---|---|
| generation | ACE Codeco eval. subset | Prolog DCG | 40.8 | 0.00286 |
| generation | ACE Codeco eval. subset | Java Earley parser | 1040. | 0.0730 |
| parsing | ACE Codeco eval. subset | Prolog DCG | 5.13 | 0.000360 |
| parsing | ACE Codeco eval. subset | Java Earley parser | 392. | 0.0276 |
| parsing | full ACE Codeco | Prolog DCG | 20.7 | 0.00146 |
| parsing | full ACE Codeco | Java Earley parser | 1900. | 0.134 |
| parsing | full ACE | APE | 230. | 0.0161 |

**Performance Tests of the Implementations.** Finally, the performance of the two implementations can be evaluated and compared. Both can be used for parsing and for generation, and thus the runtimes in these two disciplines can be compared.

The first task was to generate all sentences of the evaluation subset of ACE Codeco up to the length of seven tokens. The second task was to parse the sentences that result from the generation task. This parsing task was performed in two ways for both implementations: once using the evaluation subset and once using the full ACE Codeco grammar. The restricted lexicon of the evaluation subset was used in both cases. These tests were performed on a MacBook Pro laptop computer having a 2.4 GHz Intel Core 2 Duo processor and 2 GB of main memory. Table 1 shows the results of these performance tests.

The generation of the resulting 14'240 sentences only requires about 41 seconds in the case of the Prolog DCG implementation. This means that less than 3 milliseconds are needed on average for generating one sentence. The Java implementation, in contrast, needs about 17 minutes for this complete generation task, which corresponds to 73 milliseconds per sentence. Thus, generation is about 25 times faster when using the Prolog DCG version compared to the Java implementation. These results show that the Prolog DCG implementation is well suited for exhaustive language generation. The Java implementation is much slower but the time values are still within a reasonable range.

The Prolog DCG approach is amazingly fast for parsing the same set of sentences using the evaluation subset of the grammar. Here, parsing just means detecting that the given statements are well-formed according to the grammar. Altogether only slightly more than 5 seconds are needed to parse the complete test set, i.e. less than 0.4 milliseconds per sentence. When using the full ACE Codeco grammar for parsing the same set of sentences, altogether 21 seconds are needed, i.e. about 1.5 milliseconds per sentence. The Java implementation is again much slower and requires almost 30 milliseconds per sentence when using the grammar of the evaluation subset, which leads to an overall time of more than 6 minutes. For the full grammar, 134 milliseconds are required per sentence leading to an overall time of about 32 minutes. Thus, the Java implementation is

76 to 92 times slower than the Prolog DCG for the parsing task. Because all time values are clearly below 1 second per sentence, both parser implementations can be considered fast enough for practical applications.

The fact that the Java implementation requires considerably more time than the Prolog DCG is not surprising. DCG grammar rules in Prolog are directly translated into Prolog clauses and generate only very little overhead. Java, in contrast, has no special support for grammar rules: they have to be implemented on a higher level. The same holds for variable unifications, which come for free with Prolog but have to be implemented on a higher level in Java.

As a comparison, the existing parser APE — the reference implementation of ACE — needs about 4 minutes for the complete parsing task. Thus, it is faster than the Java implementation but slower than the Prolog DCG version of Codeco. However, it has to be considered that APE does more than just accepting well-formed sentences. It also creates a DRS representation and a syntax tree.

## 7 Conclusions

In summary, the Codeco notation allows us to define controlled natural languages in a convenient and adequate way, including complex nonlocal phenomena like anaphoric references. The resulting grammars have a declaratively defined meaning, can be interpreted in different kinds of programming languages in an efficient way, and allow for lookahead features, which are important for predictive editors. Furthermore, Codeco enables automatic grammar testing, e.g. by exhaustive language generation, which can be considered very important for the development of reliable practical applications. Altogether, Codeco embodies a more engineering focused approach to CNL.

Codeco is a proposal for a general CNL grammar notation. It cannot be excluded at this point that extensions become necessary to express the syntax of other CNLs, but Codeco has been shown to work very well for a large subset of ACE, which is one of the most advanced CNLs to date.

## References

1. Angelov, K., Ranta, A.: Implementing Controlled Languages in GF. In: Fuchs, N.E. (ed.) CNL 2009. LNCS, vol. 5972, pp. 82–101. Springer, Heidelberg (2010)
2. Chomsky, N.: On binding. Linguistic Inquiry 11(1), 1–46 (1980)
3. Clark, P., Chaw, S.-Y., Barker, K., Chaudhri, V., Harrison, P., Fan, J., John, B., Porter, B., Spaulding, A., Thompson, J., Yeh, P.: Capturing and answering questions posed to a knowledge-based system. In: K-CAP 2007: Proceedings of the 4th International Conference on Knowledge Capture, pp. 63–70. ACM (2007)
4. Clark, P., Harrison, P., Jenkins, T., Thompson, J., Wojcik, R.H.: Acquiring and using world knowledge using a restricted subset of English. In: Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2005), pp. 506–511. AAAI Press (2005)

5. Cole, R., Mariani, J., Uszkoreit, H., Varile, G.B., Zaenen, A., Zampolli, A., Zue, V. (eds.): Survey of the State of the Art in Human Language Technology. Cambridge University Press (1997)
6. Dahl, V., Tarau, P., Li, R.: Assumption grammars for processing natural language. In: Naish, L. (ed.) Proceedings of the Fourteenth International Conference on Logic Programming, pp. 256–270. MIT Press (1997)
7. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM 13(2), 94–102 (1970)
8. Fuchs, N.E., Kaljurand, K., Kuhn, T.: Attempto Controlled English for Knowledge Representation. In: Baroglio, C., Bonatti, P.A., Małuszyński, J., Marchiori, M., Polleres, A., Schaffert, S. (eds.) Reasoning Web. LNCS, vol. 5224, pp. 104–124. Springer, Heidelberg (2008)
9. Funk, A., Tablan, V., Bontcheva, K., Cunningham, H., Davis, B., Handschuh, S.: CLOnE: Controlled Language for Ontology Editing. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 142–155. Springer, Heidelberg (2007)
10. Johnson, S.C.: Yacc: Yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA (July 1975)
11. Joshi, A.K., Levy, L.S., Takahashi, M.: Tree adjunct grammars. Journal of Computer and System Sciences 10(1), 136–163 (1975)
12. Kaljurand, K.: Paraphrasing controlled English texts. In: Pre-Proceedings of the Workshop on Controlled Natural Language (CNL 2009). CEUR Workshop Proceedings, vol. 448, CEUR-WS (April 2009)
13. Kaplan, R.M., Bresnan, J.: Lexical-functional grammar: A formal system for grammatical representation. In: Bresnan, J. (ed.) The Mental Representation of Grammatical Relations, pp. 173–281. MIT Press (1982)
14. Knuth, D.E.: Backus normal form vs. backus naur form. Communications of the ACM 7(12), 735–736 (1964)
15. Kuhn, T.: How controlled English can improve semantic wikis. In: Proceedings of the Forth Semantic Wiki Workshop (SemWiki 2009). CEUR Workshop Proceedings, vol. 464, CEUR-WS (2009)
16. Kuhn, T.: Codeco: A grammar notation for controlled natural language in predictive editors. In: Pre-Proceedings of the Second Workshop on Controlled Natural Languages (CNL 2010). CEUR Workshop Proceedings, vol. 622, CEUR-WS (2010)
17. Kuhn, T.: Controlled English for Knowledge Representation. PhD thesis, Faculty of Economics, Business Administration and Information Technology of the University of Zurich (2010)
18. Kuhn, T., Schwitter, R.: Writing support for controlled natural languages. In: Proceedings of the Australasian Language Technology Association Workshop 2008, pp. 46–54 (December 2008)
19. Lappin, S., Leass, H.J.: An algorithm for pronominal anaphora resolution. Computational Linguistics 20(4), 535–561 (1994)
20. Martin, P.: Knowledge Representation in CGLF, CGIF, KIF, Frame-CG and Formalized-English. In: Priss, U., Corbett, D.R., Angelova, G. (eds.) ICCS 2002. LNCS (LNAI), vol. 2393, pp. 77–91. Springer, Heidelberg (2002)
21. Naur, P., Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Perils, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language ALGOL 60. Communications of the ACM 6(1), 1–17 (1963)

22. Pereira, F., Warren, D.H.D.: Definite clause grammars for language analysis. In: Readings in Natural Language Processing, pp. 101–124. Morgan Kaufmann Publishers (1986)
23. Pollard, C., Sag, I.: Head-Driven Phrase Structure Grammar. Studies in Contemporary Linguistics. Chicago University Press (1994)
24. Power, R., Stevens, R., Scott, D., Rector, A.: Editing OWL through generated CNL. In: Pre-Proceedings of the Workshop on Controlled Natural Language (CNL 2009). CEUR Workshop Proceedings, vol. 448, CEUR-WS (April 2009)
25. Schwitter, R., Kaljurand, K., Cregan, A., Dolbear, C., Hart, G.: A comparison of three controlled natural languages for OWL 1.1. In: Proceedings of the Fourth OWLED Workshop on OWL: Experiences and Directions. CEUR Workshop Proceedings, vol. 496, CEUR-WS (2008)
26. Schwitter, R., Ljungberg, A., Hood, D.: ECOLE — a look-ahead editor for a controlled language. In: Controlled Translation — Proceedings of the Joint Conference Combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop (EAMT-CLAW 2003), Ireland, pp. 141–150. Dublin City University (2003)