Journal of Logic, Language and Information

A Principled Approach to Grammars for Controlled Natural Languages and Predictive Editors

Tobias Kuhn

Abstract Controlled natural languages (CNL) with a direct mapping to formal logic have been proposed to improve the usability of knowledge representation systems, query interfaces, and formal specifications. Predictive editors are a popular approach to solve the problem that CNLs are easy to read but hard to write. Such predictive editors need to be able to "look ahead" in order to show all possible continuations of a given unfinished sentence. Such lookahead features, however, are difficult to implement in a satisfying way with existing grammar frameworks, especially if the CNL supports complex nonlocal structures such as anaphoric references. Here, methods and algorithms are presented for a new grammar notation called Codeco, which is specifically designed for controlled natural languages and predictive editors. A parsing approach for Codeco based on an extended chart parsing algorithm is presented. A large subset of Attempto Controlled English (ACE) has been represented in Codeco. Evaluation of this grammar and the parser implementation shows that the approach is practical, adequate and efficient.

Keywords Anaphoric references · Attempto Controlled English · Chart parsing · Controlled natural languages · Predictive editors

1 Introduction

Controlled natural languages (CNL) [40,51] are based on natural language but come with restrictions concerning lexicon, syntax and/or semantics. They have been proposed to make knowledge representations [15,42,13,44], rule systems [25,45], query interfaces [35,3,50,31], and formal specifications [14] more user-friendly. The scope of this work is restricted to CNLs with a direct and deterministic mapping to some kind of formal logic, like Attempto Controlled English (ACE) [13], Computer Processable English [47], and CLOnE [15]. The approach presented here does *not* target less restricted languages like Basic English [37], Caterpillar Fundamental English [49], or ALCOGRAM [1].

The work presented here was funded by the research grant (Forschungskredit) programs 2006 and 2008 of the University of Zurich. I would like to thank Norbert E. Fuchs, George Herson, Stefan Höfler, Michael Hess, Kaarel Kaljurand, Marc Lutz, and Adam Wyner for comments, discussions and proof-reading on this paper and earlier versions thereof.

Tobias Kuhn
Department of Pathology, Yale University School of Medicine
E-mail: kuhntobias@gmail.com
http://www.tkuhn.ch

While CNLs are easier to understand than comparable logic languages [30], they have the problem that it can be very difficult for users to write statements that comply with the syntactic restrictions of the language. Three approaches have been proposed so far to solve this problem: error messages [5,10], conceptual authoring [41,12], and predictive editors [48,43,3,27]. Each of them has its own advantages and drawbacks. Error messages are the most straightforward approach, and they leave much freedom to users. However, it is very difficult to give good and helpful error messages, because the whole richness of unrestricted language can be expected in statements written by users who are not familiar with the details of the restricted grammar. The other two approaches are based on sophisticated editing environments, which solve the problems of error messages but which also heavily confine the freedom of users during the writing process. Conceptual authoring is a top-down approach that enables users to gradually create concrete statements via given operations on incomplete sentences (i.e. sentences with "holes"). Predictive editors, in contrast, implement a left-to-right approach. They show all possible continuations of unfinished sentences (i.e. sentences that are "chopped off" at the end), enabling step-by-step creation of CNL statements. For any unfinished sentence, a predictive editor shows all possible words or phrases that can be used to continue the sentence in a grammar-compliant way. This approach seems very promising, and it is the one that is followed in the work presented here. Experiments have shown that well-designed predictive editors enable easy creation of well-formed sentences even for users unfamiliar with the respective CNL [3,27,31].

Obviously, predictive editors depend on the availability of lookahead features, i.e. the retrieval of the possible continuations to a given unfinished sentence on the basis of a given grammar. This is not a trivial task, especially if the grammar describes complex nonlocal structures like anaphoric references. Apart from that, it is desirable to have a simple grammar notation that is fully declarative and can easily be implemented in different kinds of programming languages. This is because good tool integration is crucial for the usability and acceptance of CNL approaches.

These requirements are explained in more detail in the next section, and existing grammar frameworks are assessed with respect to these requirements (Section 2). Then, the Codeco grammar notation is presented (Section 3), which is designed to solve the identified problems. Next, an implementation of this grammar notation is introduced in the form of a chart parser (Section 4), and specific applications of Codeco are discussed (Section 5), namely a concrete grammar written in Codeco and a CNL editor making use of that grammar. Finally, different aspects of Codeco and its parser are evaluated (Section 6), before we draw the conclusions (Section 7).

This paper is a revised and substantially extended version of a workshop paper [29]. For more details, see the author's doctoral thesis [28].

2 Background

To our knowledge, the Grammatical Framework (GF) [2] is the only existing grammar framework that has a specific focus on CNL. GF fulfills most but not all of the requirements that will be introduced shortly (it has no particular support for describing the resolvability of anaphoric references and does not support dynamic lexica). With extensions (perhaps inspired by Codeco), it could become a suitable grammar notation for the specific problem described in this paper.

The remainder of this section first puts forward a list of grammar requirements (Section 2.1), and then discusses different kinds of existing grammar frameworks and assesses them with respect to our requirements: natural language grammar frameworks (Section 2.2), parser generators (Section 2.3), and definite clause grammars (Section 2.4).

2.1 Grammar Requirements

The implementation of CNLs to be used in predictive editors raises a number of requirements concerning the used grammar notation. We argue for the following six requirements to enable efficient and practical CNL applications:

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. With concrete grammars, the syntax trees for given statements can be automatically built, and therefore structural ambiguity can be automatically spotted, among other things.

Declarativeness. CNL grammars should be declarative, i.e. defined in a way that does not depend on a specific algorithm or implementation. Declarative grammars can be completely separated from the parser that processes them. This makes it easy to use such grammars from 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 an unfinished sentence can be continued. For this reason, the CNL should be defined in a form that enables the efficient implementation of such lookahead features. Concretely, this means that an unfinished sentence, 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 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 possible continuations.

Anaphoric References and Scoping. The proper handling of anaphoric references is a tricky issue for predictive editors. Grammars for CNLs that support anaphoric references should describe the circumstances under which such references are allowed in an exact, declarative, and simple way. In order to have a clear separation of syntax and semantics, resolvability of anaphoric references should be treated as a syntactic issue that does not depend on the semantic representation. Concretely, a predictive editor 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. What makes things more complicated is the fact that the resolvability of anaphoric references depends on the scoping of the preceding text, triggered by negation markers and different kinds of quantifiers. While scoping in natural language can be considered a semantic phenomenon, it has to be treated as a syntactic issue in CNLs if the restrictions on anaphoric references are to be described appropriately. We use the term *anaphoric reference* in a broad sense that includes definite noun phrases like "the country".

Dynamic Lexicon. The lexicon of a CNL should be dynamic, i.e. extensible during the parsing process. This is required to make it possible for users to add new words while they are writing a sentence. Being forced to cancel the creation of a sentence just because of a missing word is very frustrating and inefficient.

Implementability. Finally, a CNL grammar notation should be simple enough to be easy to implement in different programming languages and should be neutral with respect to the programming paradigm of its parser. This 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 the fact that there can be many other tools besides the parser that need to read and process the grammar, including editors, paraphrasers, and verbalizers. Furthermore, more than one parser might be necessary for practical reasons: A simple top-down parser, for example, may be the best option when used for parsing large texts in batch mode and for doing regression tests (e.g. through language generation), but a chart parser is much better suited for providing lookahead capabilities.

2.2 Natural Language Grammar Frameworks

A large number of grammar frameworks exist for natural languages. Some of the most popular ones are *Head-driven Phrase Structure Grammars* (HPSG) [39], *Lexical-Functional Grammars* [23], *Tree-Adjoining Grammars* [22], *Generalized Phrase Structure Grammars* [16], and *Combinatory Categorial Grammars* [46], but many more exist [7]. Most of these frameworks are defined in an abstract and declarative way. Concrete grammars based on such frameworks, however, are mostly hard-coded in a certain programming language and do not have a declarative nature that would make them independent from their parsers.

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 language 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 somewhat opposing goals. Natural language grammars are *language descriptions* that describe existing phenomena. CNL grammars, in contrast, are *language definitions* that define new artificial languages, which in some sense just happen to look like natural language.

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 difficult. Another reason is that lookahead seems to be much less relevant for natural language applications, and thus no special attention has been paid 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 have emerged from an NLP background like CPL [6] and CLOnE [15] (which are otherwise very different from each other).

The handling of ambiguity is another important difference. Natural language grammars have to deal with the inherent ambiguity of natural language. 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 logic-based ones on which this paper focuses) remove ambiguity by their design, which typically makes underspecification and heuristics unnecessary.

Finally, anaphora resolution is another particularly difficult problem for the correct representation of natural language. In computational linguistics, this problem is usually solved by

applying algorithms of different degrees of sophistication to find the most likely antecedents (e.g. [19,33]). This is a difficult problem because 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 the reference. 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. This stands in contrast to the requirement introduced in Section 2.1 that a CNL should define the resolvability of anaphoric references on the syntactic level. Natural language grammar frameworks like HPSG establish *binding theories* [4,39] to address the problem of anaphoric references. These binding theories 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 [36,24]. Examples of more sophisticated grammar formalisms for formal languages — called *parser generators* — include Yacc [21] and GNU bison[1]. Formalized-English [34] is an example of a CNL defined in such a parser generator notation.

The general problem of these formalisms is that context-sensitive constraints cannot be defined in a declarative way. With plain Backus-Naur-style grammars, context-free languages can be described in a declarative and simple way, but such grammars are very limited and even very simple CNLs cannot be defined appropriately. It is possible to include context-sensitive elements, but 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 it comes to more complex languages.

When discussing lookahead capabilities, 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 difficult 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) [38] are a simple but powerful notation to define grammars for natural and formal languages. They are almost always written in logic-based programming languages like Prolog. In fact, many of the grammar frameworks for natural languages introduced above are often implemented on the basis of Prolog DCGs.

DCGs are fully declarative in their core (e.g. Prolog DCG rules that do not use curly brackets or cuts). Therefore, they can in principle be processed by any programming language. Building

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

upon the logical concept of definite clauses, they are easy to interpret for logic-based programming languages, but a considerable overhead is necessary in other programming languages to simulate backtracking and unification.

DCGs are good in terms of expressiveness because they are not necessarily context-free but can contain context-sensitive elements. Anaphoric references, however, are again a problem. It is difficult to define them in an appropriate way. 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 unifies the agreement structure of the pronoun with the first possible element of the antecedent list. This part of the code is not fully declarative. A more serious problem, however, is the way 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 grammar rule:

```
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. This kind of threading of anaphoric information is similar to an approach called *DRS threading* for the generation of semantic representations, in which case anaphoric information can be included with little overhead [20]. However, as motivated above, we would like to treat the resolution of references as a syntactic issue, independent of the semantic representation.

Different DCG extensions have been proposed in order to describe natural language in a more appropriate way. Assumption Grammars [9], for example, are motivated by natural language phenomena that are difficult to express otherwise, like free word order. Simple anaphoric references can be represented in a very clean way, but the approach does not scale up to complex anaphor types like non-reflexive pronouns.

A further problem with the DCG approach concerns lookahead features. In principle, it is possible to provide lookahead features with standard Prolog DCGs [32], but this solution is not very efficient and can become impractical for complex grammars and long sentences.

3 The Codeco Notation

This section presents a new grammar notation called *Codeco*, which stands for "*co*ncrete and *de*clarative grammar notation for *co*ntrolled natural languages". It is specifically designed for CNLs in predictive editors and should allow for relatively simple implementations of complete and correct lookahead features. While many elements of Codeco are straightforward and very similar to existing approaches, the proper handling of anaphora requires novel types of special elements in the form of forward and backward pointing references. With Codeco, anaphoric information flows through the syntax tree in the particular top-down, left-to-right path described by Johnson and Klein [20].

Below, the elements of the Codeco notation are introduced, i.e. grammar rules, grammatical categories, and certain special elements. These different elements are motivated by ACE examples. After that, the issue of reference resolution is discussed in more detail.

3.1 Simple Categories and Grammar Rules

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

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

Terminal categories are represented in square brackets:

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

There is a special notation for pre-terminal categories, which are marked with an underline:

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

Pre-terminal categories can be expanded but only to terminal categories, i.e. they 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:

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

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

In order to support context-sensitivity, non-terminal and pre-terminal categories can be augmented with flat feature structures. Feature representations use 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:

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

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

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

Feature values *cannot* be feature structures themselves, i.e. they 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 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

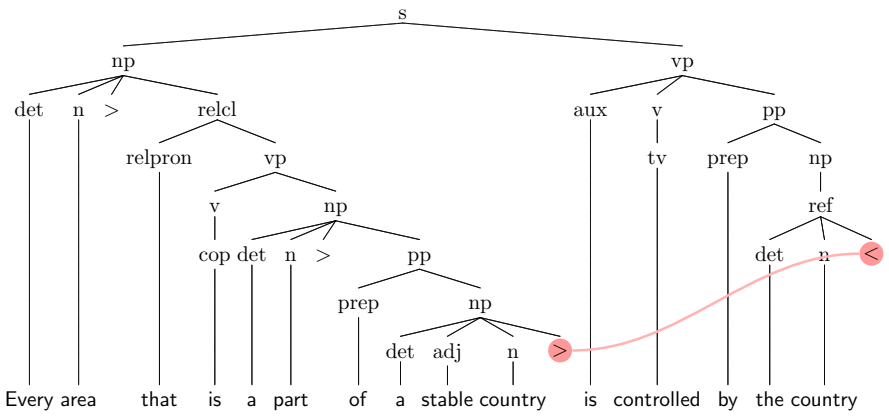
Every area that is a part of a stable country is controlled by the country.

where "the country" is a resolvable anaphoric reference (as mentioned above, our usage of the term *anaphoric reference* includes definite noun phrases). With the Codeco elements introduced so far, however, 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 picture 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:

$$np \xrightarrow{\vdots} [\,\text{a}\,]\ \ \underline{noun}\left(\text{text: } \boxed{\text{Noun}}\right)\ \ >\left(\begin{matrix}\text{type: noun}\\ \text{noun: } \boxed{\text{Noun}}\end{matrix}\right)$$

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

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.

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

3.3 Scoping and Accessibility

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

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

where one can refer to "man" or to "house" but not to "enemy" (because "every" opens a scoping that is closed after "enemy"). Therefore, "himself" and "the house" are possible continuations, but not "the enemy". The Codeco elements introduced so far do not allow for such restrictions. Additional elements are needed to define where scopings open and where they close.

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

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

Scopings have to be closed somewhere. In contrast to the opening positions of scopings, their closing positions can be far away from the scope-triggering structure. For this reason, the closing positions cannot be defined in the same way. Instead, Codeco defines 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 scoping that is opened by the direct or indirect children of "v" and "np" is closed at the end of "np". If no scopings have been opened, scope-closing rules simply behave like normal rules. See Section 3.8 for a visualization of the scoping and accessibility aspects of the example shown above. In contrast to most other approaches, Codeco defines scopings in a way that is completely independent from the semantic representation.

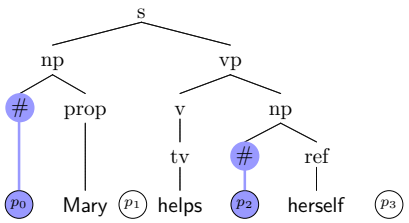
3.4 Position Operators

With the introduced Codeco elements, anaphoric definite noun phrases like "the area" can be restricted to positions where they are resolvable. At this point, however, we cannot define 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 (each p_i being a position identifier):



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:

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

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

As we will see, a further extension is needed for the appropriate definition of non-reflexive pronouns.

3.5 Negative Backward References

We need to solve a further problem, which concerns variables as they are supported by some CNLs like ACE. Phrases like "a person X" can be used to introduce a variable "X". The question is how to treat cases where 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, such that subsequent occurrences of "X" can only refer to the second but not to the first. 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 "$\not<$", which ensures 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:

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

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 Codeco elements that have been introduced are still not sufficient for expressing all the things we would like to express. As already mentioned, there is still a problem with non-reflexive pronouns like "him". While reflexive pronouns like "himself" can be restricted to refer only to the respective subject, non-reflexive pronouns cannot be prevented from doing so:

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

To distinguish such cases, it 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. This allows for correctly representing non-reflexive pronouns:

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

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 powerful constructs, which restrict anaphoric references 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: Say that "this" should be used to refer to antecedents which are either neuter and have no variable attached or which are of type "relation" (whatever that means), while in neither case being a proper name or the subject of the sentence. This complex behavior could be achieved with the following rule:

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

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. Usually, proper names are 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*:

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

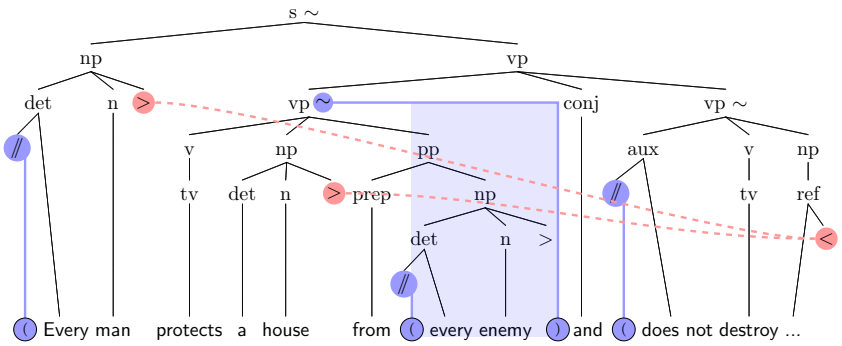
Strong forward references are always accessible even if they are within closed scopings. Apart from that, they behave like normal forward references.

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 scoping that has been closed before the position of the backward reference, or if it is a strong forward reference.

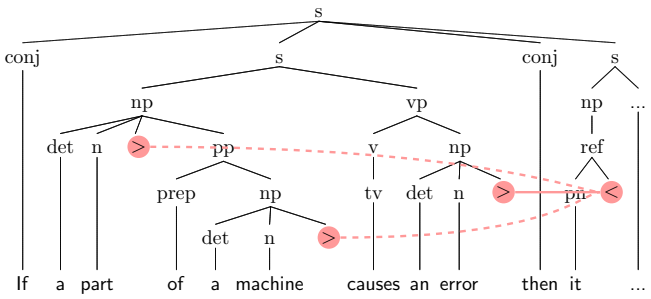
This accessibility constraint can be visualized in the syntax tree. For the unfinished sentence shown in Section 3.3, the syntax tree could look as follows:



All nodes that represent the head of a scope-closing grammar rule are marked with "∼". The positions in the text where scopings open and close are marked with parentheses. In this example, three scopings 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 scoping. 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 unfinished sentence. In contrast, the forward reference for "enemy" is not accessible because it is inside a closed scoping. The possible references are shown as dashed lines. Thus, the unfinished sentence can be continued with the anaphoric references "the man" or "the house" (or equivalently "himself" or "it", respectively) but not with 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. 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, 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:

$$\textit{ref} \;\overset{\vdots}{\longrightarrow}\; [\,\text{the}\,] \;<\!\begin{pmatrix}\text{type: noun}\\ \text{noun: } \boxed{\text{N}}\end{pmatrix} \;\; \underline{\textit{noun}}\Big(\text{text: } \boxed{\text{N}}\Big)$$

$$\textit{ref} \;\overset{\vdots}{\longrightarrow}\; [\,\text{the}\,] \;\; \underline{\textit{noun}}\Big(\text{text: } \boxed{\text{N}}\Big) \;<\!\begin{pmatrix}\text{type: noun}\\ \text{noun: } \boxed{\text{N}}\end{pmatrix}$$

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. This restriction is important for the lookahead algorithm to be presented, but it is not relevant for the parsing algorithm.

4 Codeco in a Chart Parser

In order to provide lookahead features for predictive editors, chart parsers are a good choice. In addition, they are well-suited for implementations in procedural or object-oriented programming languages, because they do not depend on backtracking. The basic idea is to store temporary parse results in a data structure that is called a *chart* and that contains small portions of parse results in the form of *edges*.

The algorithm for Codeco to be presented here is based on the chart parsing algorithm invented by Jay Earley, which is therefore known as the *Earley algorithm* [11]. Grune and Jacobs [18] discuss this algorithm in more detail, and Covington [8] shows how it can be implemented. The specialty of the Earley algorithm is that it combines top-down and bottom-up processing.

The parsing time of the standard Earley algorithm is in the worst case cubic with respect to the number of tokens to be parsed and only quadratic for the case of unambiguous grammars. However, this holds only if the categories have no arguments (e.g. feature structures). Otherwise, parsing is NP-complete in the general case. This means that for *certain* grammars, longer sentences cannot be parsed efficiently. Fortunately, grammars describing natural language typically do not fall into this worst-case category. Furthermore, there is a certain soft upper limit on the length of natural sentences (sentences of more than 1000 words are rarely found in a natural context, and only a few instances with more than 10 000 words are reported). For these reasons, Earley parsers usually perform very well for natural grammars and the majority of input texts.

The algorithm described in this section has been implemented in Java. This implementation is the basis for the predictive editor that is used in AceWiki and the ACE Editor, which will be

introduced in the next section. The code is available as open source as part of the AceWiki code base.[2]

To describe the elements of the chart parser and to explain the different steps of the parsing algorithm, the following meta language symbols will be used:

- F stands for a feature structure, i.e. a set of name/value pairs.
- A stands for any (terminal, pre-terminal or non-terminal) category, i.e. a category name followed by an optional feature structure.
- α stands for an arbitrary sequence of zero or more categories.
- r stands for a forward reference symbol, i.e. either "$>$" or "$\gg$".
- ρ stands for an arbitrary sequence of zero or more forward references "rF" and scope openers "$/\!/$".
- s stands for either a colon ":" or a tilde "$\sim$" so that "$\xrightarrow{s}$" can stand for "$\xrightarrow{:}$" or for "$\xrightarrow{\sim}$".
- i stands for a position identifier that represents a certain position in the input text.

All meta symbols can have a numerical index to distinguish different instances of the same symbol, e.g. α_1 and α_2. Other meta symbols will be introduced as needed.

Below, the chart parser elements and parsing steps are explained (Sections 4.1 and 4.2), before the actual lookahead algorithm is introduced (Section 4.3).

4.1 Chart Parser Elements

Before we can turn to the actual parsing steps, the fundamental elements of Earley parsers are introduced (chart and edges) together with a graphical notation that will be used to describe the parsing steps in an intuitive way.

Edges. Every edge of a chart parser is derived from a grammar rule and — like the grammar rule — consists of a head and a body:

$$\langle i_1, i_2 \rangle \quad A \rightarrow \alpha_1 \bullet \alpha_2$$

The body of the edge is split by the dot symbol "$\bullet$" into a sequence α_1 of categories that have already been recognized and another sequence α_2 of categories that still have to be processed. In addition, every edge has a start position i_1 and an end position i_2. Such an edge tells us that we started looking for category A at position i_1. Up to position i_2, we have found the category sequence α_1, but to complete category A, we still need to recognize the sequence α_2.

Edges where all categories of the body are recognized are called *passive*. All other edges are called *active*, and their first category of the sequence of not-yet-recognized categories is called their *active category*.

Edges with Antecedents. Processing Codeco grammars requires an extended notation for edges. Whenever a backward reference occurs, we need to be able to find out which antecedents are accessible from that position. For this reason, edges coming from a Codeco grammar have to carry information about the accessible antecedents.

First of all, every edge must carry the information whether it originated from a normal rule or a scope-closing one. Edges originating from normal rules are called *normal edges* and edges coming from scope-closing rules are called *scope-closing edges*. Like the rules they originate from, normal edges are represented by an arrow with a colon "$\xrightarrow{:}$" and scope-closing edges use an arrow with a tilde "$\xrightarrow{\sim}$".

[2] `https://github.com/AceWiki/AceWiki`

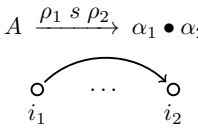
In addition, every Codeco edge has two sequences which are called *external antecedent list* and *internal antecedent list*. Both lists are displayed above the arrow: the external one to the left of the colon or tilde, the internal one to the right thereof. Both antecedent lists are sequences of forward references and scope openers. Hence, Codeco edges have the following general structure:

$$\langle i_1, i_2 \rangle \quad A \xrightarrow{\rho_1 \ s \ \rho_2} \alpha_1 \bullet \alpha_2$$

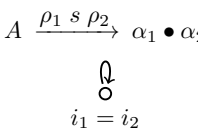
ρ_1 is the external antecedent list. It represents the antecedents that come from outside the edge, i.e. from positions earlier than i_1. ρ_2 is the internal antecedent list. It contains the antecedents that come from inside the edge, i.e. from the categories of α_1 and their children. Internal antecedents come from somewhere between the start and the end position of the respective edge. Scope openers in the antecedent lists show where scopings have been opened that are not yet closed up to the given position.

Chart. A chart is a data structure used to store the partial parse results in the form of edges. During the parsing process, edges are added to the chart, which is initially empty. Edges are not changed or removed from the chart once they are added (unless the input text changes). Traditionally, chart parsers perform a *subsumption check* for each new edge to be added to the chart [8]: A new edge is added only if no equivalent or more general edge already exists. For reasons that will become clear later, the algorithm to be presented requires an *equivalence check* instead of a subsumption check: New edges are added to the chart except for the case that there exists an edge that is fully equivalent.

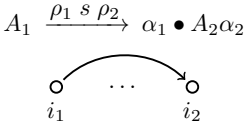
Graphical Notation. In order to be able to describe the chart parsing steps for the Codeco notation in an intuitive way, a simple graphical notation is used that is inspired by Gazdar and Mellish [17]. The positions of the input text are represented by small circles that are arranged as a horizontal sequence, and edges are represented as arrows that point from their start position to their end position having a label with the remaining edge information. This gives the following representation of a general edge:



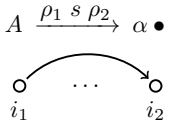
The three dots "..." mean that i_2 is either the same position as i_1 or directly follows i_1 or indirectly follows i_1. Thus, it includes the case $i_1 = i_2$ that would be represented as



Active edges can be generally represented as



where A_2 is the active category of the edge. Passive edges, in contrast, have the general form



where the dot is at the last position of the body. This graphical notation is used below to describe the parsing steps in an explicit but intuitive way.

4.2 Chart Parsing Steps

In a traditional Earley parser, there are four parsing steps: initialization, scanning, prediction and completion. In the case of Codeco, an additional step — to be called *resolution* — is needed to resolve the references, position operators, and scope openers. Below, the general algorithm is explained, each of the five parsing steps is described, and some brief complexity considerations are given.

4.2.1 General Algorithm

The general algorithm starts with the initialization step to initialize the empty chart. Then, prediction, completion and resolution are performed several times, which together will be called the *PCR* step. This PCR step corresponds to the "Completer/Predictor Loop" as described by Grune and Jacobs [18]. A text is parsed by consecutively scanning the tokens of the text and by performing the PCR step after each scanning of a token. The following piece of pseudocode shows this general algorithm:

```
parse(tokens) {
    new chart
    initialize(chart)
    pcr(chart)
    foreach t in tokens {
        scan(chart,t)
        pcr(chart)
    }
}
```

The PCR step consists of repeatedly executing the prediction, completion and resolution steps until none of the three is able to generate an edge that is not yet in the chart:

```
pcr(chart) {
    loop {
        c := chart.size()
        predict(chart)
        complete(chart)
        resolve(chart)
        if c=chart.size() then return
    }
}
```

The actual order of these three steps can be changed without breaking the algorithm, but it can have an effect on the performance.

In terms of performance, there is potential for optimization anyway. First of all, the above algorithm checks the chart for new edges only after the resolution step. An optimized algorithm checks after each step whether the last three steps contributed a new edge or not. Furthermore, a progress table can be introduced that allows the different parsing steps to remember which edges

of the chart they already checked. In this way, edges can be prevented from being checked by the same parsing step more than once. Such an optimized algorithm can look as follows (without going into the details of the progress table):

```
pcr(chart) {
    step := 0
    i := 0
    new progressTable
    loop {
        c := chart.size()
        if step=0 then predict(chart,progressTable)
        if step=1 then complete(chart,progressTable)
        if step=2 then resolve(chart,progressTable)
        if c=chart.size() then i := i+1 else i := 0
        if i>2 then return
        step := (step+1) modulo 3
    }
}
```

The variable `i` counts the number of consecutive idle steps, i.e. steps that did not increase the number of edges in the chart. The loop can be exited as soon as this value reaches 3. In this situation, no further edge can be added because each of the three sub-steps has been performed on exactly the same chart without being able to add a new edge.

4.2.2 Graphical Notation for Parsing Steps

Building upon the graphical notation for edges introduced above, the parsing steps will be described by the use of a graphical notation with a large arrow in the middle of the picture corresponding to the following scheme:

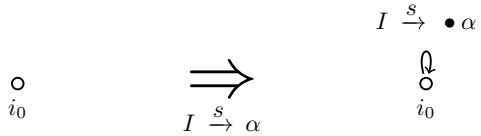
(edges in the chart) $\overset{\text{(rule in the grammar)}}{\Longrightarrow}$ (edge to be added)

On the left hand side of the arrow, edges are shown that need to be in the chart in order to execute the described parsing step. If a grammar rule is shown below the arrow then this rule must be present in the grammar for executing the parsing step. On the right hand side of the arrow the new edge is shown that has to be added to the chart when the described parsing step is executed, unless the resulting edge is already there.

If a certain meta symbol occurs more than once on the left hand side of the picture and in the rule representation below the arrow then this means that the respective parts have to be unifiable but not necessarily identical. When generating the new edge, these unifications have to be considered but the existing edges in the chart and the grammar rules remain unchanged.

4.2.3 Initialization

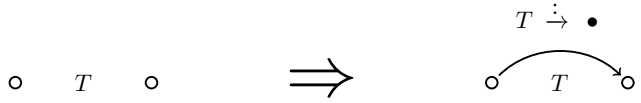
At the very beginning, the chart has to be initialized. For each rule that has the topmost category (something like "*text*" or "*sentence*") on its left hand side, an edge is introduced into the chart at the start position:



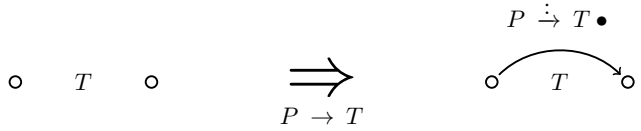
i_0 stands for the start position, i.e. the position in front of the first token, and I stands for the topmost category of the grammar. The only difference to the standard Earley algorithm is that the information about normal and scope-opening rules is taken over from the grammar to the chart, represented by s.

4.2.4 Scanning

During the scanning step, a token is read from the input text. This token is interpreted as a terminal symbol T, for which a passive edge is introduced that has T on its left hand side:



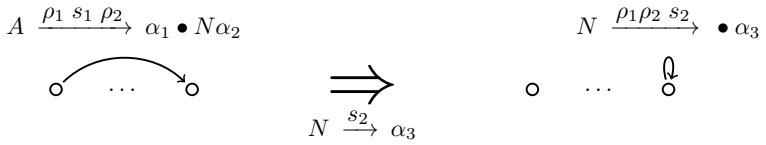
In addition, the token can occur in one or more lexical rules:



The rule $P \rightarrow T$ can come from the static grammar or from a dynamic lexicon.

4.2.5 Prediction

The prediction step looks out for grammar rules that could be applied at the given position. For every active category in the chart that matches the head of a rule in the grammar, a new edge is created:



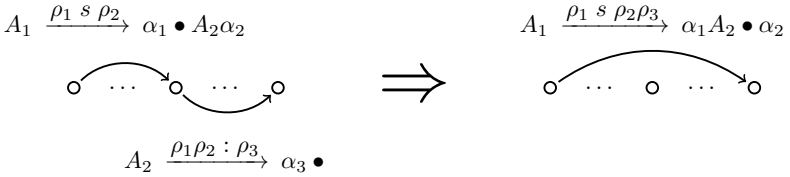
N denotes a non-terminal category. The external antecedent list of the new edge is a concatenation of the external and the internal antecedent list of the existing edge. The internal antecedent list of the new edge is empty because it has no recognized categories in its body and thus cannot have internal antecedents.

Because the two shown nodes can actually refer to the same position, the new edge that is produced by such a prediction step can itself be used to produce more new edges by again applying the prediction step at the same position.

4.2.6 Completion

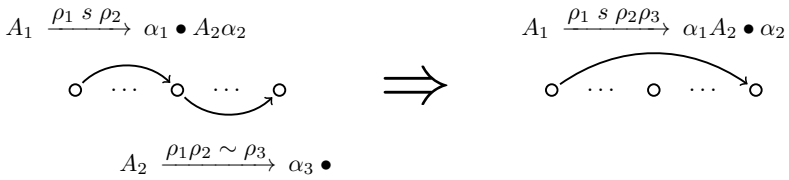
The completion step takes the active categories of the active edges in the chart and looks for passive edges with a corresponding head. If such two edges can be found then a new edge can be created out of them.

In the standard Earley algorithm, there is only one kind of completion step. In the case of Codeco, however, it is necessary to differentiate between the cases where the passive edge is a normal edge and those where it is a scope-closing edge. In the case of a normal edge, the completion step looks as follows:



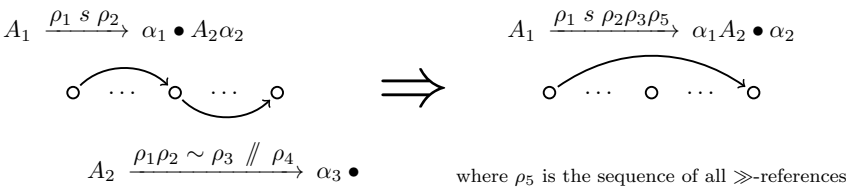
In contrast to the standard Earley algorithm, not only the active category of the active edge has to match the head of the passive edge, but also the references of the active edge have to be present in the same order in the passive edge.

If no scoping has been opened then scope-closing edges are completed in exactly the same way as normal edges:



where ρ_3 contains no scope opener $/\!/$

If one or more scopings have been opened then everything that comes after the first scope opener (except strong forward references) is removed from the internal antecedent list for the new edge to be added:

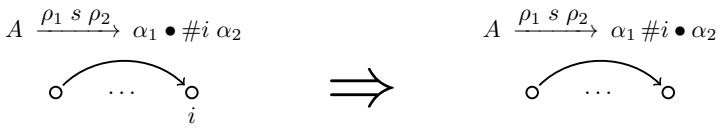


This rule ensures that references within scopings are not accessible from the outside.

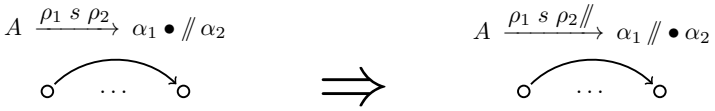
4.2.7 Resolution

In order to handle position operators, scope openers, and references, an additional parsing step is needed, which is called *resolution*. Generally, only elements occurring in the position of an active category are resolvable.

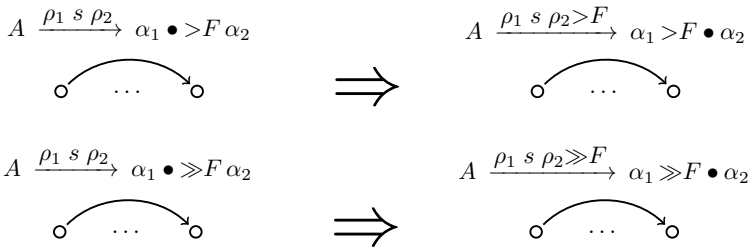
A position operator is resolved by unifying its variable with an identifier that represents the given position in the input text:



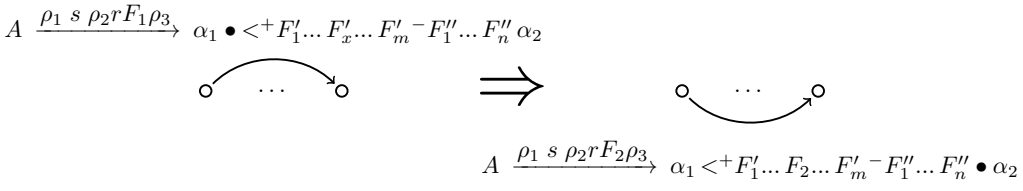
Scope openers are resolved by adding the scope opener symbol to the end of the internal antecedent list:



Forward references are resolved in a similar way. Together with their feature structure, they are added to the end of the internal antecedent list:



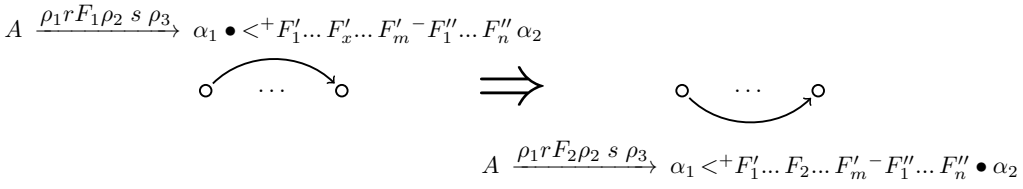
Complex backward references can be resolved to an internal antecedent or — if this is not possible — to an external one. The resolution to an internal antecedent works as follows (with $1 \leq x \leq m$ and $0 \leq n$):



where F_1 is unifiable with F'_x and is not unifiable with any F'', and where ρ_3 contains no rF_3 such that F_3 is unifiable with an F' while being not unifiable with any F''

where F_1 and F'_x are unified and F_2 is the result of this unification

The positive feature structures of the complex backward reference are denoted by F', the negative ones by F''. The resolution to an external antecedent is straightforward:

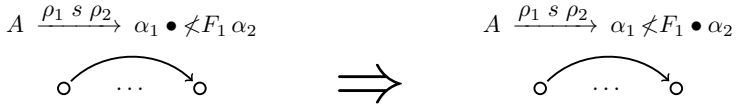


where F_1 is unifiable with F'_x and is not unifiable with any F'', and where ρ_2 and ρ_3 contain no rF_3 such that F_3 is unifiable with an F' while being not unifiable with any F''

where F_1 and F'_x are unified and F_2 is the result of this unification

Note that the same edge can produce more than one new edge when several positive feature structures can unify with the same forward reference. Since normal backward references are equivalent to complex ones for the case $x = m = 1$ and $n = 0$, they do not need to be discussed separately.

Negative backward references, finally, can be resolved only if no matching antecedent exists, neither internal nor external:



where ρ_1 and ρ_2 contain no rF_2 such that F_2 can unify with F_1

Here it becomes clear why an equivalence check — and not just a subsumption check — is needed before adding new edges to the chart. A negative backward reference that is resolvable given certain antecedent lists is not necessarily resolvable in the case of antecedent lists that are more general. More specific edges can behave differently than general ones, and for this reason an edge has to be added to the chart even if a more general edge already exists.

4.2.8 Complexity Considerations

Here, some brief and scruffy complexity considerations for the presented algorithm are given. This is done by comparing it to the standard Earley algorithm, which has been proven to be efficient in practical applications.

The space requirements of chart parsers can be measured by the size of the chart, i.e. by the number of contained edges. As long as the positive feature structures of complex backward references are pairwise disjoint (i.e. not unifiable), the special elements of Codeco increase the number of edges in the chart — compared to the standard Earley algorithm — only linearly with respect to the number of special elements used in the grammar, and only by a constant factor with respect to the length of the token list. This can be seen by the fact that the scanning, prediction, and completion steps do not produce more edges than in the standard algorithm. Furthermore, for each edge and its descendant edges that contain special elements, the resolution step can be applied at most once for each special element. The only exception are complex backward references with positive feature structures that are not pairwise disjoint. Thus, the chart can be expected to remain reasonably small as long as complex backward references with more than one positive feature structure are used with caution.

In terms of time complexity, it is easy to verify that the additional time needed — compared to the standard algorithm — for processing any edge in the prediction or resolution step or any two edges in the completion step is linear with respect to the number of elements in the external and internal antecedent list. Since the number of elements in the antecedent lists is linearly correlated with the number of parsed tokens and since the number of tokens increases the chart only by a constant factor, it can be concluded that the amount of additional time that is needed grows only in a linear way with respect to the number of tokens. Furthermore, checking for the equivalence of edges does not take more than twice as much time compared to checking for subsumption (because equivalence can be checked by a mutual check for subsumption). Altogether, the presented algorithm can be expected to be reasonably fast.

4.3 Lookahead with Codeco

Given Codeco's chart parsing algorithm, lookahead features — as they are needed for predictive editors — can be efficiently implemented in a relatively simple way. The basic idea is that the lookahead information is stored in the active categories. These are categories that are predicted to possibly occur after the end position of the respective edge. Thus, the possible next tokens can be found in the active terminal categories of the edges that have their end positions at the

end of the given unfinished sentence. Pre-terminal categories and backward references, however, make the actual algorithm a little more complicated.

The possible next tokens are described as sets of options where at least one of the options must be fulfilled by a token to be a possible continuation of a given unfinished sentence. The algorithm to be introduced can describe the possible next tokens in an abstract and in a concrete way by generating a set of abstract options O_a and another set of concrete options O_c. An abstract option would say, for example, that any proper name is a possible next token, whereas a concrete option could say that the concrete proper name "Bill" is a possible token.

In order to get this lookahead information, the unfinished sentence has to be parsed, i.e. the chart has to be filled with the edges that represent the syntactic structure of the unfinished sentence. As a next step, the abstract options can be extracted. After that, the concrete options can be created using the abstract options and the lexicon entries.

Extraction of Abstract Options. First of all, a formal structure for abstract options has to be defined. In the algorithm to be presented, abstract options have the form

$$C/\{X_1 \dots X_n\}$$

with $n \geq 0$ and where C and each X_j are terminal or pre-terminal categories. C denotes a category of possible next tokens with X_j being exceptions in the form of more specific categories describing tokens that are not possible. For instance, the abstract option

$$\underline{var} \,/\, \Big\{ \; \underline{var}\Big(\text{varname: X}\Big) \;\; \underline{var}\Big(\text{varname: Z}\Big) \Big\}$$

states that all tokens of the pre-terminal category "*var*" are possible next tokens with the exception of those with a "varname" feature value of "X" or "Z". Concretely, this means that any variable is a possible next token except "X" and "Z". Another example is

$$\underline{pron}\begin{pmatrix}\text{refl: –}\\ \text{gender: fem}\end{pmatrix} / \{\}$$

that denotes that any non-reflexive feminine pronoun is a possible next token. Terminal categories can also appear in abstract options, e.g.

$$[\,\text{that}\,] \,/\, \{\}$$

stating that the word "that" is a possible next token.

The set of abstract options O_a is extracted from the edges of the chart. This is done by iterating over all edges that have their end position at the position where the unfinished sentence ends. This position is denoted by i_x. Only edges are relevant that have a terminal or pre-terminal category (denoted by T) as their active category.

First, let us consider edges that have a complex backward reference after their active category. For every edge — and for every possible F'_x therein — of the form

$$A \xrightarrow{\rho_1 \; s \; \rho_2} \alpha_1 \bullet T{<}^{+}F'_1 \dots F'_x \dots F'_m \,{}^{-}F''_1 \dots F''_n \alpha_2$$

with $1 \leq x \leq m$ and $0 \leq n$, and for every rF_1 that is contained in ρ_1 or in ρ_2 and that has a feature structure F_1 that is unifiable with F'_x, an abstract option

$$T'/\{T''_1 \ldots T''_t\}$$

is added to O_a where T' is the result of category T after unifying F_1 and F'_x and where the exceptions are obtained as follows: For every F'' that is unifiable with F_1, an exception T'' is added that is the result of category T after unifying F_1 and F''. The differentiation between T and T' is necessary because the unification of F_1 and F'_x can entail the binding of variables that also occur in T. Altogether, this has the effect that terminal or pre-terminal categories in front of backward references are reported as possible next tokens with exceptions that describe all cases for which the reference can afterwards not be resolved. Again, this part of the algorithm described on the basis of complex backward references also applies for normal backward references, which will not be discussed separately because they are just a special case.

Next, we have to handle edges with negative backward references. For every edge of the form

$$A \xrightarrow{\rho_1\ s\ \rho_2} \alpha_1 \bullet T \not< F \alpha_2$$

an abstract option

$$T/\{T'_1 \ldots T'_n\}$$

is added to O_a where the exceptions T'_i are obtained as follows: For every rF' that is contained in ρ_1 or in ρ_2 and that has a feature structure F' that is unifiable with F, an exception T'_j is added that is the result of category T after unifying F and F'. The effect of this is that terminal or pre-terminal categories that are in front of negative backward references are reported as possible next tokens together with exceptions that describe all cases where the negative backward reference can afterwards find a matching antecedent and thus cannot be resolved.

So far, these option descriptions "look" only one step ahead. They do not cover cases where more than one terminal or pre-terminal category exists between the active position and the backward reference. In the case of normal and complex backward references, however, it is possible and useful to look more than one step ahead. The symbol δ is used to represent a sequence of one or more terminal or pre-terminal categories.

For every edge — and for every possible F'_x therein — of the form

$$A \xrightarrow{\rho_1\ s\ \rho_2} \alpha_1 \bullet T \delta <^+ F'_1 \ldots F'_x \ldots F'_m\ ^- F''_1 \ldots F''_n \alpha_2$$

and for every rF that is contained in ρ_1 or in ρ_2 and that has a feature structure F that is unifiable with F'_x, an abstract option

$$T'/\{\}$$

is added to O_a where T' is the result of category T after unifying F and F'_x.

Finally, edges that have a terminal or pre-terminal category at their active position but are not covered by the patterns introduced so far, an abstract option is created the following way: For every edge of the form

$$A \xrightarrow{\rho_1\ s\ \rho_2} \alpha_1 \bullet T\alpha_2$$

that is not covered by the patterns introduced above, an abstract option

$$T/\,\{\}$$

is added to O_a. This means that when no backward reference is close to the active category then this terminal or pre-terminal category is reported as a category of a possible next token.

In this way, a set of abstract options O_a is obtained that describes the possible next tokens in a general way, i.e. without considering the lexicon. Such general lookahead information can be important for predictive editors, e.g. for allowing users to add new words on the fly.

Extraction of Concrete Options. In contrast to abstract options, which describe possible next tokens without explicitly listing them, concrete options show the concrete terminal categories that are possible at the given position in the text.

Concrete options could actually just be terminal categories. For user-friendly predictive editors, however, it can be necessary to know the pre-terminal categories from which they are derived, e.g. for grouping the possible next words into different submenus. For this reason, concrete options have the form

$$W \leftarrow C$$

where W is a terminal category and C is a pre-terminal category from which W has been derived. The following example represents the possibility to continue the unfinished text with the noun "country":

$$[\,\text{country}\,] \;\leftarrow\; \underline{\textit{noun}}\Big(\text{human:}-\Big)$$

The special symbol "$\varnothing$" is used at the position of C if the given word does not originate from a lexical rule but directly from the grammar. The concrete option

$$[\,\text{every}\,] \;\leftarrow \varnothing$$

for instance, states that "every" is a possible next token that does not come from the lexicon but is part of the grammar rules.

The set of concrete options O_c is generated on the basis of the abstract options O_a. For every abstract option

$$W/\,\{\}$$

that is contained in O_a and where W is a terminal category, a concrete option

$$W \leftarrow \varnothing$$

is added to O_c. In addition, for every abstract option

$$C/\,\{X_1 \ldots X_n\}$$

where C is a pre-terminal category, and for each lexical rule

$$C' \;\rightarrow\; W$$

where C' is unifiable with C but is not unifiable with any X_j, the concrete option

$$W \leftarrow C$$

is added to O_c.

In this way, we obtain a set of concrete options O_c containing the concrete word forms that are possible to follow the given unfinished sentence.

Lookahead Interface. Parsers that implement these algorithms can provide a simple interface for predictive editors to access the lookahead features. Assuming that the unfinished text has been submitted to the parser, the predictive editor module can simply request the set of concrete options O_c and — if needed — the set of abstract options O_a. In this way, the predictive editor module has all needed information in order to show how the text can be continued, e.g. in the form of graphical menus.

The set of concrete options can directly be presented to the user as possible next words. On the basis of the set of abstract options, the predictive editor can, for example, allow users to create new words that are not yet known at the time the lookahead algorithm runs. Thus, the predictive editor does not only know which concrete words are possible at the given position but also which words in general would be allowed if they were in the lexicon.

This kind of lookahead is not necessarily restricted to the level of individual words. Tokens can be multi-word units such as "it is false that". Moreover, the introduced lookahead algorithm can be applied several times to find sequences of two, three or more tokens to continue the given unfinished sentence. A predictive editor could show a selection of these, e.g. the most frequently used ones. The predictive editor implementation to be introduced shortly does not currently support this kind of multi-token lookahead, but there are no technical obstacles to implementing it with the presented techniques.

Completeness and Correctness. The presented lookahead algorithm can be analyzed with respect to completeness and correctness: It is complete in the sense that it returns every token for which, together with the tokens of the unfinished sentence, a *complete* syntax tree exists that is well-formed according to the grammar. It is correct in the sense that it only returns the tokens for which, together with the tokens of the unfinished sentence, a *partial* syntax tree exists that is well-formed and does not end with an unresolvable reference.

These definitions of completeness and correctness leave some freedom on how to handle certain special cases. They do not say anything about the tokens that lead to a well-formed partial syntax tree that cannot be completed to a full statement. This can happen, for example, if the edge used for predicting the next token contains at a later position a non-terminal category that does not occur as a head in any of the grammar rules. In this case, the edge can never complete and the predicted token is actually not a possible next token to complete the unfinished sentence, even though a well-formed partial syntax tree can be constructed.

A stronger correctness criterion that requires the existence of a completion for the partial syntax tree seems not practical, because its detection would entail a complex search problem on the grammar rules. Thus, Codeco grammars should be designed in such a way that invalid statements fail at the earliest possible position, i.e. at the first position for which no continuation to a well-formed statement exists. It could be argued that properly designed grammars should follow this restriction anyway.

5 Applications

This section briefly introduces two concrete applications of the Codeco notation: the ACE Codeco grammar and the ACE Editor. As illustrated in Figure 1, these two applications are on different conceptual levels. The ACE Editor depends on the ACE Codeco grammar and on the Codeco chart parser, which all depend on the Codeco notation.

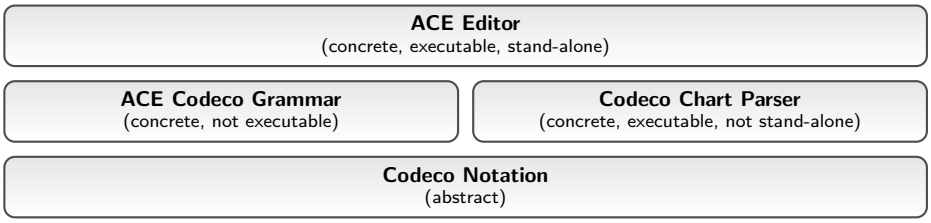


Fig. 1 The dependencies of the different components (higher components depend on lower ones). The attributes in parentheses emphasize their different types.

5.1 ACE Codeco Grammar

The ACE Codeco grammar is — as its name suggests — a grammar in the Codeco notation describing a subset of the language ACE. It consists of 164 grammar rules[3] and 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 supported in the form of simple definite noun phrases, variables, and reflexive and non-reflexive 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 nonlocal phenomena like anaphoric references.

5.2 ACE Editor

The ACE Editor[4] is a general web-based editor for writing and modifying ACE texts. Its purpose is to demonstrate how user interfaces can be designed and used to write and modify CNL texts in a simple and intuitive way. Users should not need to learn the grammar of ACE in advance, but they should be able to learn the language while using the editor. A predictive editor is included that uses the ACE Codeco grammar introduced above. Different representations for ACE sentences can be shown like syntax trees, paraphrases and logical formulas, which are all generated by the ACE parser. Figure 2 shows a screenshot with the predictive editor as an internal window. The predictive editor is a modular component that can be reused by other applications. In fact, AceWiki [26,27] and Coral [31] use this predictive editor. No reasoning features are implemented at this point, but applications like AceWiki demonstrate how reasoning can be tightly integrated in such systems.

Obviously, the predictive editor is the most important part of the ACE Editor. In designing predictive editors, the biggest challenge is arguably the fact that there can be a large number of possible ways to continue an unfinished sentence. All these possibilities should be shown to the users in a simple and understandable way, allowing them to quickly choose the option they are looking for. To solve this problem, the ACE Editor uses menu boxes that occupy most of

[3] See [28] for the complete grammar.

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

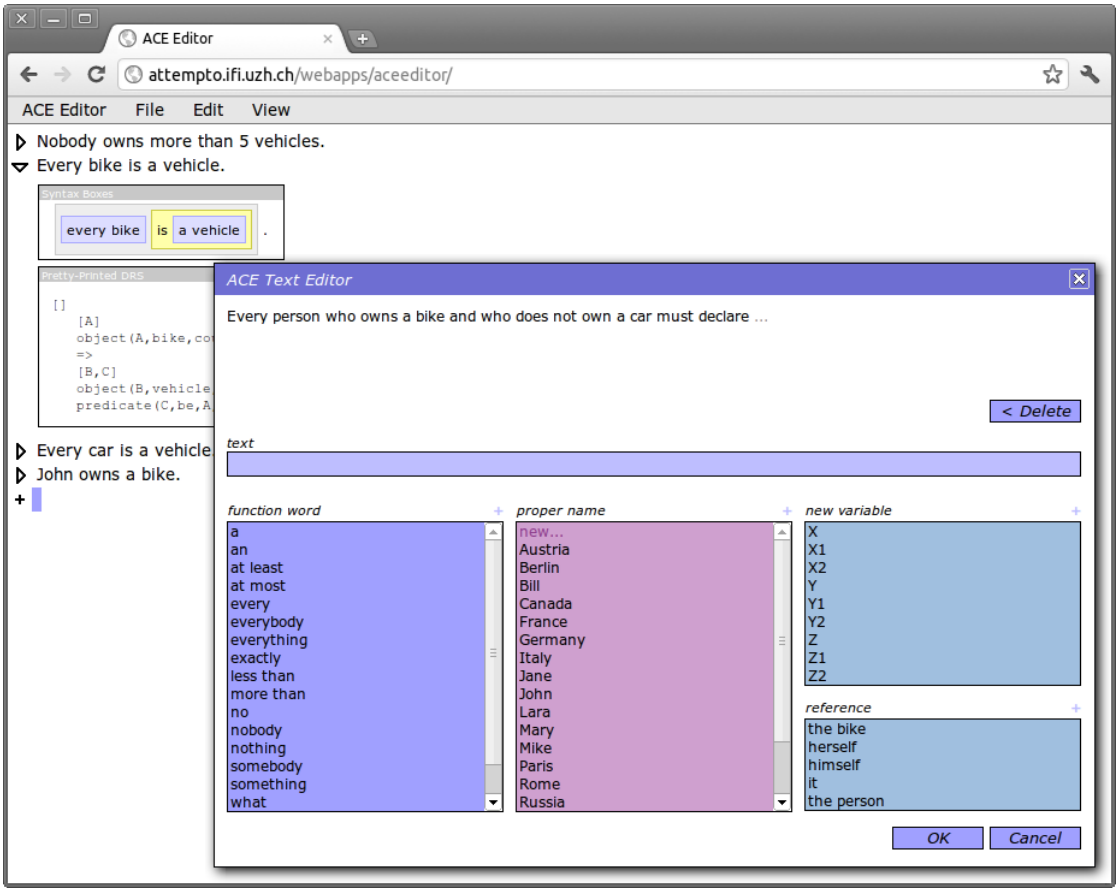


Fig. 2 The ACE Editor with its predictive editor.

the space of the predictive editor window. Each of these boxes contains the menu items for a particular type of word, showing scroll bars when the given vertical space is not sufficient. The text field above the menu boxes can be used to filter the items. In this way, large numbers of possible next tokens can be presented to the users, and the selection of an item only requires one mouse click and possibly some scrolling or filtering.

The unfinished sentence shown in the predictive editor in Figure 2 demonstrates the capability of Codeco to correctly predict resolvable anaphoric references. For the given unfinished sentence, the editor suggests "the bike" and "the person" but not "the car", because "car" is under the scope of the negation expressed with "does not" and is therefore not accessible.

6 Evaluation

Different aspects of the approach have to be evaluated. Basically, all four components shown in Figure 1 need scrutiny. The top component has been evaluated in our previous work: The usability of the ACE Editor's predictive editor has been tested in several end user experiments. The results showed that it is effectively usable by untrained participants [27,31]. Below, evaluation results on the two middle components are presented: the ACE Codeco grammar and the chart parser.

The bottom component — the Codeco notation — is an abstract one and does therefore not allow for direct evaluation, but we can get indirect results from evaluating the concrete grammar and the parser.

The evaluation presented here relies on the method of exhaustive language generation, i.e. the generation of all possible sentences from a given grammar and lexicon up to a certain fixed sentence length. The main problem of this approach is that one quickly encounters combinatorial explosion on the number of generated sentences. In practice, this means that one can only use a subset of the grammar. 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. The lexicon of this subset only contains one entry per word category: the proper name "Mary", the noun "woman", the adjective "young", the transitive adjective (i.e. a combination of adjective and preposition) "mad-about", the intransitive verb "wait", the transitive verb "ask", the adverb "early", the preposition "for", the variable "X", and the number "2".

We start with evaluating the ACE Codeco grammar. There are two properties we can test: (1) Is the language described by ACE Codeco unambiguous, and (2) is it indeed a subset of ACE?

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. Exhaustive language generation operates on the syntax trees of sentences, which means that ambiguous sentences are generated several times with different syntax trees attached. Looking for duplicates in the list of generated sentences reveals whether there is ambiguity or not (for the given grammar, lexicon, and sentence length).

Up to the length of ten tokens, the evaluation subset of ACE Codeco generates 2 250 869 sentences. If sorted alphabetically, these are the first and last sentences:

a woman asks a woman .
a woman asks a woman and asks a woman .
a woman asks a woman and asks everybody .
a woman asks a woman and asks everybody early .
...
X waits for X for X for who ?
X waits for X for X for who early ?
X waits for X for X for X .
X waits for X for X for X early .

Due to the restricted lexicon, many of these sentences are repetitive. One should bear in mind that exhaustive language generation is not about generating natural sentences, but just syntactically well-formed ones. As it turns out, all generated sentences are distinct, which — by design — means that each has only one syntax tree. Thus, at least a large subset of ACE Codeco (its evaluation subset) is unambiguous for at least relatively short sentences (up to ten tokens).

Subset Check of ACE Codeco and Full ACE. The ACE Codeco grammar is designed as a proper subset of ACE. We can now check whether this is the case, again for the evaluation subset and up to a certain sentence length.

Every sentence up to the length of ten tokens — as shown above — 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.

Table 1 The results of a performance test of the two implementations of Codeco, with the existing ACE parser (APE) for comparison (all numbers are rounded to three significant digits).

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

Next, we can evaluate the parser implementation. There exists a second implementation of Codeco based on a simple transformation into Prolog DCGs [28], which can be used for comparison. This Prolog DCG implementation supports all Codeco elements, but does not implement lookahead features. We can therefore test the following two properties: (1) Are the two implementations equivalent, i.e. do they return the same results for given grammars, and (2) how fast are they compared to each other and compared to the ACE parser?

Equivalence Check of the Implementations. Both Codeco implementations (chart parser and Prolog DCG) support exhaustive language generation. This allows us to check whether the two implementations accept the same set of sentences, as they should. We take the ACE Codeco grammar to run this comparison.

Generating all sentences of ACE Codeco up to eight tokens results in identical sets of sentences for the two implementations. This is an indication that they contain no major bugs and interpret Codeco grammars in the same way.

Performance Tests of the Implementations. Finally, we can look at the performance of the two implementations. 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 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.

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 chart parser implementation 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 chart parser implementation is much slower but the time values are still within a time range that is more than reasonable.

The Prolog DCG approach is very 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 chart parser 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 chart parser 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 chart parser implementation in Java 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 chart parser but slower than the Prolog DCG. However, it has to be considered that APE does more than just accepting well-formed sentences: It also generates logical representations and syntax trees.

In summary, the ACE Codeco grammar can be considered unambiguous and fully ACE-compliant. The chart parser implementation is stable and reasonably fast (even though much slower than a Prolog DCG). From this we can conclude that the Codeco notation is suitable for describing controlled languages like ACE. In addition, it shows that Codeco meets the implementability requirement introduced in Section 2.1.

7 Conclusions

With Codeco, controlled natural languages of the unambiguous type can be defined in a simple and convenient way. With the help of different kinds of forward and backward references, complex nonlocal structures like anaphoric references can be defined in a fully declarative way. Following a chart parsing approach, the possible continuations of unfinished sentences can be reliably retrieved, which facilitates the implementation of predictive editors. Implementing efficient parsers for Codeco is not difficult and does not depend on a particular programming paradigm. In addition, the presented approach enables automatic grammar testing, e.g. by exhaustive language generation, which can be considered very important for the development of reliable practical applications.

Codeco can be conceived as a proposal for a general CNL grammar notation. It is possible that extensions become necessary to cover other controlled natural languages, 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. Adriaens, G., Schreors, D.: From COGRAM to ALCOGRAM: Toward a controlled English grammar checker. In: Proceedings of the 14th Conference on Computational Linguistics, vol. 2, pp. 595–601. Association for Computational Linguistics, Morristown, NJ, USA (1992)
2. Angelov, K., Ranta, A.: Implementing controlled languages in GF. In: Proceedings of the Workshop on Controlled Natural Language (CNL 2009), *Lecture Notes in Computer Science*, vol. 5972, pp. 82–101. Springer (2010)

3. Bernstein, A., Kaufmann, E.: GINO — a guided input natural language ontology editor. In: The Semantic Web — ISWC 2006, Proceedings of the 5th International Semantic Web Conference, *Lecture Notes in Computer Science*, vol. 4273, pp. 144–157. Springer (2006)
4. Chomsky, N.: On binding. Linguistic Inquiry **11**(1), 1–46 (1980)
5. 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 '07: Proceedings of the 4th International Conference on Knowledge Capture, pp. 63–70. ACM (2007)
6. 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)
7. 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)
8. Covington, M.A.: Natural Language Processing for Prolog Programmers. Prentice Hall, Englewood Cliffs, NJ, USA (1994)
9. Dahl, V., Tarau, P., Li, R.: Assumption grammars for processing natural language. In: L. Naish (ed.) Proceedings of the Fourteenth International Conference on Logic Programming, pp. 256–270. MIT Press (1997)
10. Dimitrova, V., Denaux, R., Hart, G., Dolbear, C., Holt, I., Cohn, A.G.: Involving domain experts in authoring owl ontologies. The Semantic Web — Proceedings of the 7th International Semantic Web Conference (ISWC 2008) pp. 1–16 (2008)
11. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13**(2), 94–102 (1970)
12. Franconi, E., Guagliardo, P., Tessaris, S., Trevisan, M.: Quelo: an ontology-driven query interface. In: Proceedings of the 24th International Workshop on Description Logics (DL 2011) (2011)
13. Fuchs, N.E., Kaljurand, K., Kuhn, T.: Attempto Controlled English for knowledge representation. In: Reasoning Web — 4th International Summer School 2008, *Lecture Notes in Computer Science*, vol. 5224, pp. 104–124. Springer (2008)
14. Fuchs, N.E., Schwertel, U., Schwitter, R.: Attempto Controlled English - not just another logic specification language. In: Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation (LOPSTR '98). Springer-Verlag (1990)
15. Funk, A., Tablan, V., Bontcheva, K., Cunningham, H., Davis, B., Handschuh, S.: CLOnE: Controlled language for ontology editing. In: Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference (ISWC 2007 + ASWC 2007), *Lecture Notes in Computer Science*, vol. 4825, pp. 142–155. Springer (2007)
16. Gazdar, G.: Generalized Phrase Structure Grammar. Harvard University Press (1985)
17. Gazdar, G., Mellish, C.: Natural Language Processing in PROLOG. Addison-Wesley (1989)
18. Grune, D., Jacobs, C.J.: Parsing Techniques — A Practical Guide, second edn. Monographs in Computer Science. Springer Science+Business Media, New York, NY, USA (2008)
19. Hobbs, J.R.: Resolving pronoun references. Lingua **44**(4), 311–338 (1978)
20. Johnson, M., Klein, E.: Discourse, anaphora and parsing. In: Proceedings of the 11th coference on Computational linguistics, COLING '86, pp. 669–675. Association for Computational Linguistics, Stroudsburg, PA, USA (1986)
21. Johnson, S.C.: Yacc: Yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA (1975)
22. Joshi, A.K., Levy, L.S., Takahashi, M.: Tree adjunct grammars. Journal of Computer and System Sciences **10**(1), 136–163 (1975)
23. Kaplan, R.M., Bresnan, J.: Lexical-functional grammar: A formal system for grammatical representation. In: J. Bresnan (ed.) The Mental Representation of Grammatical Relations, pp. 173–281. MIT Press (1982)
24. Knuth, D.E.: Backus normal form vs. backus naur form. Communications of the ACM **7**(12), 735–736 (1964)
25. Kuhn, T.: AceRules: Executing rules in controlled natural language. In: Web Reasoning and Rule Systems — First International Conference (RR 2007), *Lecture Notes in Computer Science*, vol. 4524, pp. 299–308. Springer (2007)
26. Kuhn, T.: AceWiki: A natural and expressive semantic wiki. In: Proceedings of the Fifth International Workshop on Semantic Web User Interaction (SWUI 2008) — Exploring HCI Challenges, *CEUR Workshop Proceedings*, vol. 543. CEUR-WS (2009). URL `http://ceur-ws.org/Vol-543/kuhn_swui2008.pdf`
27. 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)
28. Kuhn, T.: Controlled English for knowledge representation. Ph.D. thesis, Faculty of Economics, Business Administration and Information Technology of the University of Zurich (2010)
29. Kuhn, T.: Codeco: A practical notation for controlled English grammars in predictive editors. In: Proceedings of the Second Workshop on Controlled Natural Language (CNL 2010), *Lecture Notes in Computer Science*, vol. 7175, pp. 95–114. Springer (2012)
30. Kuhn, T.: The understandability of OWL statements in controlled English. Semantic Web journal (to appear)
31. Kuhn, T., Höfler, S.: Coral: Corpus access in controlled language. Corpora **7**(2) (2012, to appear)

32. Kuhn, T., Schwitter, R.: Writing support for controlled natural languages. In: Proceedings of the Australasian Language Technology Association Workshop 2008, pp. 46–54 (2008)
33. Lappin, S., Leass, H.J.: An algorithm for pronominal anaphora resolution. Computational Linguistics **20**(4), 535–561 (1994)
34. Martin, P.: Knowledge representation in CGLF, CGIF, KIF, Frame-CG and Formalized-English. In: Conceptual Structures: Integration and Interfaces — Proceedings of the 10th International Conference on Conceptual Structures (ICCS 2002), *Lecture Notes in Artificial Intelligence*, vol. 2393, pp. 77–91. Springer (2002)
35. Mueckstein, E.M.: Controlled natural language interfaces: the best of three worlds. In: CSC '85: Proceedings of the 1985 ACM thirteenth annual conference on Computer Science, pp. 176–178. ACM (1985)
36. 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)
37. Ogden, C.K.: The A B C of Basic English (in Basic). No. 43 in Psyche Miniatures General Series. K. Paul, Trench, Trubner, London (1932)
38. 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)
39. Pollard, C., Sag, I.: Head-Driven Phrase Structure Grammar. Studies in Contemporary Linguistics. Chicago University Press (1994)
40. Pool, J.: Can controlled languages scale to the Web? In: Proceedings of the 5th International Workshop on Controlled Language Applications (CLAW 2006) (2006)
41. 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 (2009)
42. 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)
43. 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-CLAW03), pp. 141–150. Dublin City University, Ireland (2003)
44. Shiffman, R.N., Michel, G., Krauthammer, M., Fuchs, N.E., Kaljurand, K., Kuhn, T.: Writing clinical practice guidelines in controlled natural language. In: Proceedings of the Workshop on Controlled Natural Language (CNL 2009), *Lecture Notes in Computer Science*, vol. 5972, pp. 265–280. Springer (2010)
45. Spreeuwenberg, S., Anderson Healy, K.: SBVR's approach to controlled natural language. In: Proceedings of the Workshop on Controlled Natural Language (CNL 2009), *Lecture Notes in Computer Science*, vol. 5972, pp. 155–169. Springer (2010)
46. Steedman, M., Baldridge, J.: Combinatory categorial grammar. In: Non-Transformational Syntax, pp. 181–224. Wiley-Blackwell (2011)
47. Sukkarieh, J.Z., Pulman, S.G.: Computer Processable English and McLogic. In: Proceedings of the Third International Workshop on Computational Semantics, pp. 367–380 (1999)
48. Tennant, H.R., Ross, K.M., Saenz, R.M., Thompson, C.W., Miller, J.R.: Menu-based natural language understanding. In: Proceedings of the 21st annual meeting on Association for Computational Linguistics, pp. 151–158. Association for Computational Linguistics (1983)
49. Verbeke, C.A.: Caterpillar fundamental English. Training & Development Journal **27**(2), 36–40 (1973)
50. Würsch, M., Ghezzi, G., Reif, G., Gall, H.C.: Supporting developers with natural language queries. In: ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 165–174. ACM, New York, NY, USA (2010)
51. Wyner, A., Angelov, K., Barzdins, G., Damljanovic, D., Davis, B., Fuchs, N., Hoefler, S., Jones, K., Kaljurand, K., Kuhn, T., Luts, M., Pool, J., Rosner, M., Schwitter, R., Sowa, J.: On controlled natural languages: Properties and prospects. In: Proceedings of the Workshop on Controlled Natural Language (CNL 2009), *Lecture Notes in Computer Science*, vol. 5972, pp. 281–289. Springer (2010)