Codeco: A Grammar Notation for Controlled Natural Language in Predictive Editors

Tobias Kuhn

Department of Informatics
University of Zurich

Second Workshop on Controlled Natural Language
15 September 2010
Marettimo (Italy)
Introduction

- **Problem:** Existing grammar frameworks do not work out particularly well for CNLs.
- **Reason:**
  - CNLs have essential differences to other languages (natural and formal ones)
  - To solve the writability problem, CNLs have to be embedded in special tools with very specific requirements.
    - Error messages and suggestions
    - Predictive editors
    - Language generation
- **Solution:** A new grammar notation that is dedicated to CNLs and predictive editors.
CNL Grammar Requirements

**Concreteness:** CNL grammars should be fully formalized and interpretable by computers.

**Declarativeness:** CNL grammars should not depend on a concrete algorithm or implementation.

**Lookahead Features:** CNL grammars should allow for the retrieval of possible next tokens for a partial text.

**Anaphoric References:** CNL grammars should allow for the definition of nonlocal structures like anaphoric references.

**Implementability:** CNL grammars should be easy to implement in different programming languages.

**Expressivity:** CNL grammars should be sufficiently expressive to express CNLs.
Lookahead Features

Predictive editors need to know which words can follow a partial text:
Anaphoric References

- Anaphoric references in CNLs are resolvable in a deterministic way:
  
  A country contains an area that is not controlled by the country. If a person X is a relative of a person Y then the person Y is a relative of the person X.
  
  John protects himself and Mary helps him.

- Anaphoric references that cannot be resolved should be disallowed:
  
  * Every area is controlled by it.
  * The person X is a relative of the person Y.

- Scopes have to be considered too:
  
  Every man protects a house from every enemy and does not destroy ...
  
  ... himself.
  
  ... the house.
  
  * ... the enemy.
Existing Grammar Frameworks

- Grammar Frameworks for Natural Languages
  - Head-Driven Phrase Structure Grammars
  - Lexical-Functional Grammars
  - Tree-Adjoining Grammars
  - Combinatory Categorial Grammars
  - Dependency Grammars
  - ... and many more

- Backus-Naur Form (BNF)

- Parser Generators
  - Yacc
  - GNU Bison
  - ANTLR

- Definite Clause Grammars (DCG)

- Grammatical Framework (GF)
## How Existing Grammar Frameworks Fulfill our Requirements for CNL Grammars

<table>
<thead>
<tr>
<th>Feature</th>
<th>NL</th>
<th>BNF</th>
<th>PG</th>
<th>DCG</th>
<th>GF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concreteness</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Declarativeness</td>
<td>+/–</td>
<td>+</td>
<td>–</td>
<td>(+)</td>
<td>+</td>
</tr>
<tr>
<td>Lookahead Features</td>
<td>–</td>
<td>+</td>
<td>(+)</td>
<td>(+)</td>
<td>+</td>
</tr>
<tr>
<td>Anaphoric References</td>
<td>(+)</td>
<td>–</td>
<td>–</td>
<td>(+)</td>
<td>–</td>
</tr>
<tr>
<td>Implementability</td>
<td>–</td>
<td>+</td>
<td>–</td>
<td>–</td>
<td>?</td>
</tr>
<tr>
<td>Expressivity</td>
<td>+</td>
<td>–</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
The Codeco Notation

Codeco = “Concrete and Declarative Grammar Notation for Controlled Natural Languages”

- Formal and Declarative
- Easy to implement in different programming languages.
- Expressive enough for common CNLs.
- Lookahead features can be implemented in a practical and efficient way.
- Deterministic anaphoric references can be defined in an adequate and simple way.
Grammar Rules in Codeco

- Grammatical categories with flat feature structures
- Category Types:
  - non-terminal (e.g. \(vp\))
  - pre-terminal (e.g. \(noun\))
  - terminal (e.g. [does not ])
- Grammar Rule Examples:
  - \(vp(\text{num: Num }, \text{neg: Neg}) \rightarrow v(\text{num: Num }, \text{neg: Neg }, \text{type: tr}) np(\text{case: acc})\)
  - \(v(\text{neg: + }, \text{type: Type}) \rightarrow [does \ not \ ] verb(\text{type: Type})\)
  - \(np(\text{noun: Noun}) \rightarrow [a] noun(\text{text: Noun})\)
  - \(noun(\text{text: woman }, \text{gender: fem}) \rightarrow [woman]\)
Forward and Backward References

The special categories “>” and “<” can be used to establish nonlocal dependencies, e.g. for anaphoric references:

\[
np \quad \xrightarrow{\cdot} \quad \text{det} \left( \text{def: -} \right) \quad \text{noun} \left( \text{text: Noun} \right) \quad > \left( \text{type: noun} \quad \text{noun: Noun} \right)
\]

\[
\text{ref} \quad \xrightarrow{\cdot} \quad \text{det} \left( \text{def: +} \right) \quad \text{noun} \left( \text{text: Noun} \right) \quad < \left( \text{type: noun} \quad \text{noun: Noun} \right)
\]
Scopes

- Opening of scopes:
  - Scopes in (controlled) English usually open at the position of the scope triggering structure, or nearby.
  - Scope opener category “∥” in Codeco:
    
    \[ quant(\text{exist: –}) \rightarrow \parallel \text{ [every]} \]

- Closing of scopes:
  - Scopes in (controlled) English usually close at the end of certain structures like verb phrases, relative clauses, and sentences.
  - Scope-closing rules “\(\sim\rightarrow\)” in Codeco:
    
    \[ \text{vp}(\text{num: Num}) \sim \rightarrow \text{v}(\text{num: Num, type: tr}) \rightarrow \text{np}(\text{case: acc}) \]
Position Operators

- How to define reflexive pronouns like "herself" that can only attach to the subject?
- With the position operator "#", position identifiers can be assigned:

\[
np(id: Id) \rightarrow \#Id \begin{array}{c}
prop(gender: G) \\
\end{array} > \begin{array}{c}
(id: Id \\
gender: G \\
type: prop)
\end{array}
\]

\[
ref(subj: Subj) \rightarrow [herself] \begin{array}{c}
(id: Subj \\
gender: fem)
\end{array}
\]

![Diagram of sentence structure](image)
Negative Backward References

- How to define that the same variable can be introduced only once?
  - *A person X knows a person X.*
- Negative backward references “≮” succeed only if no matching antecedent is accessible:

\[ \text{newvar} \rightarrow \text{var} \left( \text{text: } \square \right) \not\prec \left( \text{type: var} \right) \succ \left( \text{type: var} \right) \]
Complex Backward References

• How to define pronouns like “him” that cannot attach to the subject?
  *John helps him.*

• Complex backward references “<+...−...” have one or more positive feature structure “+” and zero or more negative ones “−”.

• They succeed if there is an antecedent that matches one of the positive feature structures but none of the negative ones:

  \[
  \text{ref} \left( \begin{array}{l}
  \text{subj: Subj} \\
  \text{case: acc}
  \end{array} \right) \rightarrow \left[ \text{him} \right] \left< + \left( \begin{array}{l}
  \text{human: +} \\
  \text{gender: masc}
  \end{array} \right) \\ 
  \left( \begin{array}{l}
  \text{id: Subj}
  \end{array} \right) \right>
  \]

• A more complicated (but probably less useful) example:

  \[
  \text{ref} \left( \begin{array}{l}
  \text{subj: Subj}
  \end{array} \right) \rightarrow \left[ \text{this} \right] \left< + \left( \begin{array}{l}
  \text{hasvar: –} \\
  \text{human: –}
  \end{array} \right) \left( \begin{array}{l}
  \text{type: relation}
  \end{array} \right) \\ 
  \left( \begin{array}{l}
  \text{id: Subj}
  \end{array} \right) \left( \begin{array}{l}
  \text{type: prop}
  \end{array} \right) \right>
  \]
Strong Forward References

• How to define that propernames like “Bill” are always accessible?
  *Mary does not love a man. Mary hates him.
  Mary does not love Bill. Mary hates him.
• Strong forward references “≫” are always accessible:

$$
np\left( \text{id: Id} \right) \rightarrow prop\left( \begin{array}{c}
\text{human: H} \\
\text{gender: G}
\end{array} \right) \gg \begin{array}{c}
\text{id: Id} \\
\text{human: H} \\
\text{gender: G} \\
\text{type: prop}
\end{array}
$$
Reference Resolution: Accessibility

Forward references are only accessible if they are not within a scope that has already been closed before the position of the backward reference:
Strong forward references are always accessible:
If a backward reference matches more than one forward reference then the closest one is taken:
Possible Extensions

- Semantics (e.g. with $\lambda$-DRSs)
- General feature structures (instead of flat ones)
- ...
Parsers for Codeco

Two parsers with different parsing approaches exist:

- **Transformation into Prolog DCG**
  - fast (1.5 ms per sentence)
  - no lookahead features
  - ideal for regression tests and parsing of large texts in batch mode

- **Execution in a chart parser (Earley parser) under Java**
  - slower, but still reasonably fast (130 ms per sentence)
  - lookahead features
  - ideal for predictive editors in Java
ACE in Codeco

- Large subset of ACE in Codeco
  - **Includes**: 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, questions, and anaphoric references (simple definite noun phrases, variables, and reflexive and irreflexive pronouns)
  - **Excludes**: 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
- 164 grammar rules
- Used in the ACE Editor:
  http://atempto.ifi.uzh.ch/webapps/aceeditor/
Evaluation of ACE Codeco

Exhaustive Language Generation:

- Evaluation subgrammar with 97 grammar rules
- Minimal lexicon
- 2,250,869 sentences with 3–10 tokens:

<table>
<thead>
<tr>
<th>sentence length</th>
<th>number of sentences</th>
<th>growth factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>87</td>
<td>14.50</td>
</tr>
<tr>
<td>5</td>
<td>385</td>
<td>4.43</td>
</tr>
<tr>
<td>6</td>
<td>1,959</td>
<td>5.09</td>
</tr>
<tr>
<td>7</td>
<td>11,803</td>
<td>6.03</td>
</tr>
<tr>
<td>8</td>
<td>64,691</td>
<td>5.48</td>
</tr>
<tr>
<td>9</td>
<td>342,863</td>
<td>5.30</td>
</tr>
<tr>
<td>10</td>
<td>1,829,075</td>
<td>5.33</td>
</tr>
<tr>
<td>3–10</td>
<td>2,250,869</td>
<td></td>
</tr>
</tbody>
</table>

- All are accepted by the ACE parser
  → ACE Codeco is a subset of ACE
- None is generated more than once
  → ACE Codeco is unambiguous
Evaluation of the Codeco notation and its implementations

Prolog DCG representation versus Java Earley parser:

• Equivalence of the Implementations:
  Generate the same set of sentences up to 8 tokens
  → The two implementations process Codeco in the same way

• Performance Tests:

<table>
<thead>
<tr>
<th>task</th>
<th>grammar</th>
<th>implementation</th>
<th>seconds/sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td>generation</td>
<td>ACE Codeco eval. subset</td>
<td>Prolog DCG</td>
<td>0.00286</td>
</tr>
<tr>
<td>generation</td>
<td>ACE Codeco eval. subset</td>
<td>Java Earley parser</td>
<td>0.0730</td>
</tr>
<tr>
<td>parsing</td>
<td>ACE Codeco eval. subset</td>
<td>Prolog DCG</td>
<td>0.000360</td>
</tr>
<tr>
<td>parsing</td>
<td>ACE Codeco eval. subset</td>
<td>Java Earley parser</td>
<td>0.0276</td>
</tr>
<tr>
<td>parsing</td>
<td>full ACE Codeco</td>
<td>Prolog DCG</td>
<td>0.00146</td>
</tr>
<tr>
<td>parsing</td>
<td>full ACE Codeco</td>
<td>Java Earley parser</td>
<td>0.134</td>
</tr>
<tr>
<td>parsing</td>
<td>full ACE</td>
<td>APE</td>
<td>0.0161</td>
</tr>
</tbody>
</table>
Conclusions

Codeco ...

• ... fulfills our requirements for CNLs in predictive editors.
• ... is suitable to describe a large subset of ACE.
• ... allows for automatic tests.

• ... stands for a principled and engineering focused approach to CNL.
Thank you for your attention!

Questions & Discussion