1 Introduction

Realization is the process to generate a sentence in human language from a logical form. It is considered the opposite of natural language understanding. In natural language understanding, the system is necessary to disambiguate the input sentence to produce the semantic representation. However in natural language realization, the system needs to decide how to a concept into words.

Realization plays an important role in many NLP applications, such as machine translation and question-answering system. In machine translation, the sentence realization generates a target-language sentence from a semantic representation obtained from the analysis and transfer processes. In question-answering system, the sentence realization generates query results in form of natural language that are summarized from many sources.

This paper presents a realizer package and its application programming interface. The realizer proposed here is developed by SWI-Prolog, a dialect of Standard Prolog.

The rest of the paper is organized as follows. Section 2 illustrates the system overview. Section 3 shows the version history of the package. Section 4 shows available predicates. Section 5, Section 6, and Section 7 describe the details of each predicate. Finally, Section 8 shows a demonstration of the realizer.

2 System Overview

The realizer consists of three modules: namely, rule compilation, lexical enumeration, and chart-based realization. The system is illustrated in Figure 1.
First, a user specifies a lexical dictionary containing list of lexicons, syntactic categories, and semantic-interpretation constraints. All lexicons are loaded into the memory and compiled into internal mathematical-representation. Once the compilation has completed, the realizer is ready to use.

After rule compilation, the user enters a semantic representation conforming to the semantic-interpretation constraints in which the dictionary postulates. The main source of semantic representation is the parser, since it analyzes a given sentence into a corresponding semantic representation. Obtaining a semantic representation, the realizer enumerates all lexicons from it and tries to reorder them to form a grammatical sentence. Parsing chart is made use in this step to check if the produced sentences and the semantic representations correspond with each other. Every possible lexicon ordering is constructed and represented by chart. Then each possibility is analyzed into semantic representation and compared with the given one. If they equal, the current lexicon ordering is returned to the user.

This realizer package provides necessary operations for semantic realization. Rules can be loaded by the predicate `load_rules/1`. The predicate `compile_rules/0` is activated within `load_rules/1` to compile all loaded lexicon rules into the internal mathematical representations. Once the rules are loaded, the user can verify what have been loaded by showing all rules with `show_rules/0`. If the user would like to clear all rules before loading a new sets of rules, he can use the predicate `clear_rules/0`. The user can realize a sentence from a semantic representation by `realize/2`. After realization, the user can also investigate the realization process on the chart by `show_chart/0`.

### 3 Version History

Version 0.1 November 4, 2006
- First released

Version 0.2 November 22, 2006
- Now supports coordination with n-ary annotation.
4 Available Predicates

- load_rules/1: This predicate loads all lexicon definition rules from the specified file and compiles them into internal representation.
- compile_rules/0: This predicate compiles all loaded lexicon definition rules.
- show_rules/0: Show all rules in the knowledge base.
- clear_rules/0: Clear all rules from the knowledge base.
- realize/2: Realize a sentence from a logical form.
- show_chart/0: Show the realization chart.

5 Grammar Rule Manipulation

5.1 Predicate: load_rules(+RuleFile)

This predicate loads all lexicon definition rules from the specified file RuleFile and compiles them into internal representation.

The algorithm of this predicate is described as follows.
1. Remove all existing lexicon definition rules from the knowledge base.
2. Load lexicon definition rules from the specified file.
3. Compile them into internal representation by calling compile_rules/0.

In this program, a lexicon definition rule is represented as \( W := C \), where \( W \) is a lexicon's surface form and \( C \) is a CCG syntactic category. Compilation of the rule is in the form of \( \text{rule}(W, R) \), where \( R \) is an internal representation of \( C \) provided by calling \( \text{ccg:conv_expr_repr}(C, R) \).

5.2 Predicate: compile_rules

This predicate compiles all loaded lexicon definition rules.

In this program, a lexicon definition rule is represented as \( W := C \), where \( W \) is a lexicon's surface form and \( C \) is a CCG syntactic category. Compilation of the rule is in the form of \( \text{rule}(W, R) \), where \( R \) is an internal representation of \( C \) provided by calling \( \text{ccg:conv_expr_repr}(C, R) \).

5.3 Predicate: compile_rule(+NumericFlag, +Category, –Result)

This predicate converts a syntactic category Category into its corresponding internal representation Result. The flag NumericFlag signifies the rule compiler to produce a special form for number handling.

5.4 Predicate: interpret_rule(+Word, +Repr, –GenRule)

Interpret a lexical rule whose surface form is Word and whose syntactic category's internal representation is Repr and yield out the generation rule in Rule.
The generation rule is in the form \( \text{rule}(\text{LF}, \text{GenPattern}) \). \text{LF} is a typed logical form (see \text{numerate_lexicons}/2 for details). \text{GenPattern} is a generation pattern which is a list of the following metarules:

1. \( \text{word}(S, R) \): a word whose surface form is \( S \) and whose syntactic category's internal representation is \( R \); \( \text{word}(\text{john}, \text{np(\text{john})}) \), for instance.

2. \( \text{leaf}(L, E) \): a branch to be recursively numerated. \( L \) represents a list of typed logical forms excluded from numeration. For example, \( \text{leaf}(\text{s(buy(A, B)}), \text{[n(A)])} \) excludes numeration of \( A \) from numeration of \( \text{s(buy(A, B}) \).

3. \( \text{coor}(C, T, A, L, R) \): a coordination. \( C \) is a conjunction name. \( T \) is a type of each argument in a list of arguments \( A \). \( L \) and \( R \) are lists of common arguments (in typed logical forms) from the left and the right sides, respectively. For example, \( \text{coor}(\text{and, s}, \text{[buy(i, coke)}, \text{drink(i, coke)}], \text{[np(i)]}, \text{[np(coke)]}) \).

The generation pattern plays an important role in the numeration process (see \text{numerate_lexicons}/2 for details).

5.5 Predicate: \text{show_rules}

Show all rules in the knowledge base.

5.6 Predicate: \text{clear_rules}

Clear all rules from the knowledge base.

6 Chart-based Realization

6.1 Predicate: \text{numerate_lexicons}(+\text{TypedLF}, –\text{Result})

List all lexicons from the specified typed logical form \text{TypedLF} and yield out the result in \text{Result}.

A typed logical form is a logical form annotated with an atomic syntactic category. For example, a typed logical form \( \text{s(buy(i, candy}) \) contains a syntactic category \( s \) and a logical form \( \text{buy(i, candy)} \).

The result returned is a list of \( \text{word}(S, I) \), where \( S \) is a surface form and \( I \) is an internal representation of the word's syntactic category, or \( \text{conj}(C) \) where \( C \) is a conjunction name. \( C \) will be comma if the conjunction is repeated. The result will be used to generate a sentence by the minimalist merging operation.

Algorithm:

1. If \text{TypedLF} is stated excluded in the exclusion list, then return an empty list. (This means no string is numerated.)

2. Otherwise, interpret each metarule of the generation pattern. The interpretation will be collected in the numeration list.
   a. \( \text{word}(S, R) \): Copy this word to the numeration list.
b. \textit{leaf}(L, E): Recursively numerate words from L with the union of the TypeLF’s exclusion list and its exclusion list. These exclusion lists are unioned to avoid repeated numeration.

c. \textit{coor}(N, T, A, L, R): numerate the coordination. The numeration of coordination is a concatenation of $Q(L), Q'(A), Q(R)$, where $Q(L), Q(R)$ are numerations of the left and the right contexts $L$ and $R$, $Q'(A)$ is a concatenation of the numeration of each argument of $A$ typed with the type $T$, with the conjunction $N$ as the separator. The conjunction is in the form of either $\text{conj}(N)$ or $\text{conj}(\text{comma})$ with respect to the repetition avoidance criteria.

3. Flatten the result.

Example:

The numeration process of $\text{np(newspaper/\relc(read(i, newspaper)))}$ is exemplified as follows.

1. Exclusion list is initially empty.

2. The generation pattern is first retrieved:

\begin{verbatim}
[ leaf(np(newspaper), []),
  word(that, ...),
  leaf(s(read(i, newspaper), [np(newspaper)])) ]
\end{verbatim}

The syntactic category's internal representation is displayed as '...

3. Interpret each metarule in the generation pattern.

1.1 \textit{leaf}(np(newspaper), []): numeration of $\text{np(newspaper)}$ is recursively done. Since the exclusion list is empty, the recursive exclusion list is also empty. Finally, the numeration of $\text{np(newspaper)}$ yields $[\text{word(newspaper, ...)}]$.

2.2 \textit{word}(that, ...): this yields $[\text{word(that, ...)}]$.

3.3 \textit{leaf}(s(read(i, newspaper)), [np(newspaper)]): The exclusion list turns to $[\text{np(newspaper)}]$. The numeration of $\text{s(read(i, newspaper))}$ is recursively done as follows.

3.3.1 The generation pattern is retrieved:

\begin{verbatim}
[ leaf(np(i), []),
  word(read, ...),
  leaf(np(newspaper), [] )
\end{verbatim}

3.3.2 Each metarule is interpreted. Finally, the numerations yield $[ [\text{word(i, ...)}], \text{word(read, ...)},
\text{word(newspaper, ...)]}$.

4.4 Finally, the numeration yields:

\begin{verbatim}
[ [\text{word(newspaper, ...)}],
  \text{word(that, ...)},
  [\text{word(i, ...)}, \text{word(read, ...)]} ]
\end{verbatim}

4. Flatten the numeration.
6.2 Predicate: realize(+LF, –Sentence)

Realize a sentence Sentence from a logical form LF.

The algorithm of this predicate is as follows.

1. Numerate lexicons from the logical form. This might generate different sets of lexicons. For each set, do 2-6.
2. Clear the realization chart.
3. Initialize the chart with the lexicons.
4. Iterate the chart building up constituencies from these lexicons. The construction is controlled by the embedded syntactic category's internal representation and the CCG's combinatory operation.
5. Find an edge whose length equals which of the number of lexicons and its logical form equals to the input one. For each edge found, do 6.
6. Yield out each resulted sentence.

6.3 Predicate: show_chart

Show the realization chart.

6.4 Predicate: clear_chart

Clear the realization chart.

6.5 Predicate: init_chart(+Words, –NoWords)

Initialize the realization chart with a list of numerated lexicons Words. The number of lexicons is yielded out in NoWords.

For each word, an edge is added to the chart. An edge is defined as edge(L, W, R) where L is the length of the edge, W is a list of surface words contained, and R is a syntactic category's internal representation. Each word in W is indexed with the position in the lexicon list Words to avoid repeated usage of lexicons.

6.6 Predicate: iterate_chart(+Len, +NoWords)

Iterate all edges in the chart, merge them, and combine their category with the specified length Len and repeat iteration with increase lengths until the length equates the specified limit NoWords.

6.7 Predicate: find_mergeable_edges(+Len, –Edge)

Find all mergeable edge pairs; the length of their concatenation equates Len. The combination of them yields Edge.
6.8 Predicate: find_conjoinable_edges(+Len, –Edge)

Find all edges conjoined with a conjunction; the length of their combination is Len. The combination yields out in Edge.

6.9 Predicate: unindex_sentence(+IndexedSentence, -Sentence).

Remove all indices from a realized sentence IndexedSentence and yield out the result in Sentence.

7 Miscellaneous Utilities

7.1 Predicate: append_lists(+Lists, –Result)

Append each list in the specified list Lists and yield out the result in Result.

7.2 Predicate: concat_lists(+Separator, +Lists, –Result)

Concatenate each list in the list Lists with a separator Separator and yield out the result in Result.

8 Example Usage

First, load the combinatory categorial grammar library to the memory.

1 ?- [ccg].
% ccg compiled into ccg 0.00 sec, 27,984 bytes
Yes

Then load the realizer module.

2 ?- [realizer].
% fs compiled into fs 0.00 sec, 3,248 bytes
% realizer compiled into realizer 0.00 sec, 18,440 bytes
Yes

At this step, we can load up the grammar rules from a file. In this example, we load up a dependency grammar for English.

3 ?- realizer:load_rules(dep_eng).
% dep_eng compiled into realizer 0.00 sec, 2,112 bytes
Yes

After loading, we can list available grammar rules.
We can realize a sentence from a logical form with the predicate \texttt{realizer:realize/2}.

\begin{verbatim}
4  ?- realizer:show_rules.
Rule: \texttt{i} := n(i)
Rule: \texttt{we} := n(we)
Rule: \texttt{they} := n(they)
Rule: \texttt{you} := n(you)
Rule: \texttt{kissed} := s(_G426 > (kiss < _G430)) \# n(_G426) /^ n(_G430)
Rule: \texttt{love} := s(_G426 > (love < _G430)) \# n(_G426) /^ n(_G430)
Rule: \texttt{will} := s(will > _G424) /^ s(_G424)
Rule: \texttt{await} := s(_G426 > (await < _G430)) \# n(_G426) /^ n(_G430)
Rule: \texttt{and} := \texttt{conj and}
Rule: \texttt{or} := \texttt{conj or}
Rule: \texttt{,} := \texttt{conj comma}
\end{verbatim}

\begin{verbatim}
5  ?- realizer:realize(s(coor(and, s, [[i > (kiss < you)], [they > (await < you)]], [], [n(you)])), S).
S = [i, kissed, and, they, await, you] ;
\end{verbatim}

No

After parsing, we can trace back realizing steps by investigating the chart.
Yes