Software Tools for Role and Reference Grammar

Chris Wilson

Incomplete Draft 1, 2006/08/06.

This paper describes the needs of linguists working in Role and Reference Grammar theory for an integrated software toolbox to assist them with studies of their languages and testing of grammatical theories. A proposed system which fulfils these needs is detailed, and progress towards the implementation of the system is explained.

Brief

Linguists working in Role and Reference grammar, like all scientists, have various needs for information storage, processing and retrieval. Likewise, they could benefit from electronic information processing tools to automate some of the repetitive manual tasks required by their work, and to help them organise, present and share their work efficiently.

An incomplete list of tools which are currently used by linguists would contain:

- Microsoft Word or a text editor, for storage of corpus text;
- Shoebox, for first-level syntactic analysis (parsing);
- (morphological analysis);
- Linguistic Tree Constructor, for creation of presentable RRG syntax trees.

Many tools could be added to this collection, to help with other repetitive and time-consuming tasks, for example:

- Annotation of corpus with high-level syntactic information from automatic and manual analysis;
- Querying and searching the corpus for high-level structures;
- Storage and maintenance of the associated lexicon for the language;
- Creation of template logical structures for lexical units, that semantically describe their meaning;
- Evaluation of template logical structures into logical statements that mathematically describe the semantic content of the text;
- Sharing information between colleagues working independently.

The present paper describes a project to develop an integrated toolbox which offers all these features. The feature list is not complete, and suggestions from linguists working in RRG and other theories are welcome. The aim of the project is to combine all the tools needed by a working linguist into a single software package that will make their work easier.

An additional benefit to linguists using this package is that their data will be uniformly coded, and comparable and electronically shareable with other linguists. Storage of this data in a central repository would be a significant step towards automated translation between languages, long promised by the theory of Role and Reference Grammar but so far remaining far out of reach. The ultimate goal of this project is to implement the first automated translation between languages using Role and Reference Grammar.

Requirements

In order to replace the collection of disparate software tools currently used by linguists, the proposed package will have to offer significant advantages. The authors feel that the combination of useful tools in the new package, fully integrated with each other, and easily available at low cost, would offer the necessary advantages in time saving and convenience to linguists to persuade them to invest their time in learning the new package, and converting their existing data.

The requirements for the package are as follows:

- Support for multiple languages;
- Storage of corpus text, and easy searching and retrieval;
- Support for multiple corpi in each language;
- Storage of structured annotations, including morphological and syntactic analysis, to the corpus text, and easy searching and retrieval;
- Automatic generation of structured annotations by parsing;
- Storage and editing of parser rules;
- Easy experimentation with new parser rules, and examination of the impact of rule changes;
- Storage and editing of the lexicon for each language;
- Assistance with entering semantic information into the lexicon;
- Connecting the lexicon to the corpus for display and editing of structured semantic information;
- Evaluation of the actual semantics (logical structures) of structures in the corpi;
- Assistance for collaboration between researchers working at remote locations;
- Low cost availability of the software package;
- Flexibility for future development through open source code and well-structured code;
- Reliability through unit and integration testing.

Design Decisions

Possible means to satisfy each requirement are analysed, and a method is chosen for each.

Support for Multiple Languages

Storage of characters from any language on a computer requires the definition of a character set, which maps characters (glyphs or shapes) to binary codes for storage.

In the past, many disparate character sets were used, some overlapping or conflicting, sometimes with incompatible alternatives for the same language. This collection of characters sets could best be described as an absolute mess, and prevented and resisted the exchange of information between computers with different language settings (character sets) for many years.

Recently, a group of experts has developed a single character set called Unicode, which includes almost every known language and character in the world. The aim was to include every language and character, but some may have been omitted due to lack of knowledge or standardisation. We can expect the omissions to be corrected in subsequent versions of the standard.

The advantages of using Unicode over separate character sets to encode a corpus include that the character set for a corpus need not be decided in advance; it is possible to mix multiple languages in a corpus; it is easy to exchange information in different languages between computers, and between

software programs on the same computer; the binary encoding is reasonably efficient and not difficult to implement.

One disadvantage is that Unicode is not yet as well supported as the old language-specific character sets, due to its young age. Some programs do not support Unicode at all, or only support some of the possible encodings. It is possible to convert between Unicode and the language-specific character sets, and the conversion for most popular character sets is already implemented and hence easily available. Support for Unicode is expected to improve over time.

Another disadvantage of Unicode is that fonts for display of Unicode characters are less widely available. Without the fonts, they will not be able to display the characters properly, although the computer will not lose the original character information when loading, saving, copying and pasting text, and it may still be possible to enter characters in the desired language. For example, SIL as only recently released free Unicode fonts for Hebrew and Ancient Greek, and Unicode Bangla fonts are not yet available for free. Probably there are languages for which no Unicode fonts exist, only the old language-specific ones. Availability of Unicode fonts, for purchase or free download,.is expected to increase over time, and the proportion of linguists who have these fonts installed on their computers is also expected to increase.

A third disadvantage is that Unicode encoding may be less efficient, and use more disk space and memory, than language-specific character sets. This is inevitable due to the flexibility of the Unicode character set, and the choices made by the designers of Unicode to favour ease of interpretation and programming over maximal space efficiency. In the light of the low and falling cost of memory and disk space, and the quantity of textual information that are likely to be processed by the software package, the authors do not consider this disadvantage to be significant.

No alternative to Unicode, as the universal character set for computers, exists at this time that is known to the authors. The decision therefore comes down to Unicode versus individual language-specific character sets. The advantages of Unicode over those character sets are considered to outweigh the disadvantages, and Unicode is chosen.

Storage of Corpus Text

Linguists will often work with a corpus of text in the languages that they study, since it provides a useful test bed for theories about the language. The corpus is traditionally stored as a flat text file, with characters in sequence, separated by spaces into words. This approach is simple and works well for small corpi, but it is not efficient to search larger corpi in this way, nor to exchange revisions to the corpus with other researchers.

A Danish researcher, Ulrik Petersen, built on the work of Crist-Jan Doedens to develop a database specifically for storing corpi and structural analysis of their contents. This database is called Emdros, and it offers a number of advantages over traditional relational databases. Some of these will concern us in the next section, but for now it suffices to say that Emdros makes it significantly easier to navigate, browse and modify structured text than either flat files or relational databases. The extent of this is shown by the fact that Ulrik is now paid a license fee by a major manufacturer of bible software to use Emdros as the underlying database of their software products. Emdros is the only database software known by the authors to be available and designed specifically for storage of corpi in any language.

The main disadvantage of Emdros from a user's point of view is that it does not provide simple graphical tools to allow users to enter and query their data. Thus, working with Emdros is currently significantly harder than working with a flat text file. In order to choose Emdros as the underlying database for this project, it would therefore be necessary to develop a simple user interface to make using Emdros at least as easy as using a word processor, while also having the advantages of the powerful structured data storage and query tools that it provides.

The authors expect that at least some linguists using this software will want to work with reasonably large quantities of text, and therefore that the advantages of using Emdros outweigh the

disadvantages. They also hope that the user interface to Emdros that will be developed as part of this project will be a useful tool in its own right.

Support for Multiple Corpi

This requirement is not difficult to achieve using any corpus database, whether flat files, Emdros or a relational database. Emdros has the notion of separate, independent databases within the same engine, which will be used to separate the corpi.

Storage of Structured Annotations

In his introduction to Emdros, Ulrik Petersen wrote: “As (Abeillé, 2003) points out, 'corpus-based linguistics has been largely limited to phenomena that can be accessed via searches on particular words. Inquiries about subject inversion or agentless passives are impossible to perform on commonly available corpora'. Emdros is a text database engine which attempts to remedy this situation in some measure. Emdros' query language is very powerful, allowing the kind of searches which Abeillé mentions to be formulated quickly and intuitively. Of course, this presupposes a database which is tagged with the data necessary for answering the query.”

For example, a linguist might want to examine all occurrences of the form “John (*verb*) (*noun*) an apple”, where (*verb*) and (*noun*) could have any value, in their corpus, in order to extract patterns or examine an odd behaviour. Traditional text searches of this form are difficult to construct and unlikely to uncover all forms in a large corpus.

In order to facilitate users in working with large corpi of text and performing complex analysis of that text, including its structure, it seems essential to provide a mechanism to store and query the structural information. For this purpose, Emdros is the only existing tool of which the authors are aware. Using Emdros for this purpose has no significant disadvantage over flat text files, where the necessary information can only be stored by defining an arbitrary, artificial and fragile syntax in the text file, which makes it significantly harder to use as a corpus. In addition, searching for complex structures that were not indexed in advance in a flat file would be virtually impossible.

Emdros allows the definition of arbitrary objects with arbitrary properties that are attached to the text at any point, even spanning multiple units of length or discontinuous blocks. Objects can represent morphemes, words, phrases, clauses, sentences, or anything at all. The types of data that can be attached include strings of arbitrary length, integer numbers, and binary data such as pictures and sounds. We do not intend to offer all of this flexibility in a user interface, but rather to make the most common and useful structures easy to enter and work with.

Automatic Generation of Structured Annotations

Having decided that we wish to store structured annotations, such as grammatical information, we require a way to input this data. Manual input must be supported, but is time-consuming and error-prone. Most structural data can be gathered by using a parser to analyse clauses automatically. The analysis will not always be correct, but can be improved by analysing failure cases and modifying the rule set.

A simple LALR parser does not cope well with all languages that must be supported. For example, many RRG users work with languages where word order is free to some extent, and simple parsers do not support this. Also, in RRG not every word (or morpheme) is part of the layered structure of the clause, since many words (and morphemes) belong in the operator projection instead. Simple LALR parsers do not provide the necessary flexibility to build two trees in this manner.

Brian Nolan has proposed that a parser based on the computer science concept of unification may be able to eliminate some false matches (over-generation of tree structures by the grammar) without over-complicating the rule set, by allowing matches based on properties of words, such as number and gender, which are stored in the lexicon. While this is not proven, it would be useful to have a platform for experimentation in such techniques.

While the basic ideas of most parsers are based on theories developed by Noam Chomsky, this does not imply or impose a Chomskyan view of language grammar, with its verb phrases and prepositional phrases, which are not supported by RRG. The Chomskyan theory of formal grammars can provide important insights into computational techniques for efficient parsing, and is more complete than the RRG "linking algorithm" which serves some of the same goals.

We propose the implementation of a parser based on Chomskyan ideas, which will allow greater experimentation to determine whether these grammars are in fact sufficient to parse sentences in the languages which linguists wish to study.

In order for RRG users to accept this parser and be willing to create rule sets for their languages, the software package must:

- offer a flexible parser which supports their requirements;
- allow fully manual input of logical structures if the user desires it;
- allow the automatic parsings to be overridden and modified by the user;
- make it easy to input and modify the parser rule set;
- make it easy to see the effects of any rule change across the corpus;
- support entering and editing of RRG clause templates;
- support filtering structures to eliminate those which do not match a template.

We propose a parser based on Chomskyan context-free grammars (type 2), where the rule set consists of rules whose left-hand side is a single non-terminal, and the right-hand side may be any number of terminals or non-terminals. In addition, the right-hand side may contain optional elements (which are allowed to be skipped), repeating elements (which may be present multiple times) and permutable elements (which may occur in any order).

Elements may specify properties which are to be retrieved from the lexicon and must match for a given word, such as number and gender. The match may be explicit, with a given value, for example a rule which matches only plural nouns. It may instead be implicit, requiring a correspondence with another element in the same rule, such as two symbols having the same value for a property such as gender. The left-hand side may specify that some of these properties of the elements are inherited up the tree, so that a noun phrase containing a plural noun becomes a plural noun phrase, and similarly singular nouns and noun phrases.

This design is experimental, and we hope to improve it based on feedback from RRG users, or abandon it when a better design becomes obvious.

Storage and Editing of Parser Rules

Having decided that a parser is necessary, and that it requires a rule set, we must now implement a system for storing and maintaining the rules.

One possibility, proposed by Brian Nolan, is that they may be stored in the lexicon. This may require an unusual structure of the lexicon, since the rules are quite different to the words (terminals) which the lexicon would normally contain. We will cover the lexicon of words (dictionary) shortly, but suffice it to say for now that the dictionary may contains the following information about each word:

- surface form(s) (as seen in the corpus);
- part of speech (noun/verb/preposition/operator);
- canonical form;
- glosses (in one or more languages);

- number and gender (may depend on the surface form);
- thematic role(s) such as *actor*, *patient*, *consumer*;

While, on the other hand, the rule set may contain the following information about each rule:

- left-hand side (non-terminal);
- right-hand side (list of non-terminals, with information about optionality, repeatability and permutability).

The only obvious correspondence between these is that the left-hand side of a parser rule may be seen as equivalent to the surface form, or the canonical form, of a word, since the words are also loaded into the parser rule set in order for the parser to work on the corpus.

However, it is possible to unify them further, if we specify that certain properties of rules are used to carry the lexical information which is to be stored in the dictionary. We give an example using the standard Backus-Naur Form (BNF), which is most frequently used to describe rule sets in text:

- *left-hand side ::= right-hand side*

where *right-hand side* consists of *terminals* (in quote marks) and *non-terminals* (unquoted). The semantic distinction is that the corpus is made up entirely of terminals, whereas non-terminals are structural (conceptual) units. For example, a rule which states that the English word “watch” is a verb would be given as follows:

- verb ::= “watch”

This states that wherever the word “watch” appears in the corpus, we may hypothesis the presence of a verb. Another rule, which might also appear in the same rule set, offers the alternative hypothesis that “watch” is actually a noun:

- noun ::= “watch”

And a rule which says that a *core* is a *predicate* followed by two *arguments* would be given as follows:

- core ::= nucleus argument argument

The syntax is extended by special notation which implies the optionality, repeatability or permutability of some of its elements. This syntax need not be described here.

We now extend the notation to allow the left-hand side to specify properties of the non-terminals which it defines, and define the following properties:

- *gender* may be *M*, *F*, or *N*;
- *number* may be *ONE* or *MANY*;
- *person* may be *FIRST*, *SECOND* or *THIRD*

The properties and their values may also be listed in the rule set, thus:

- gender ::= M
- gender ::= F
- gender ::= N
- number ::= ONE
- number ::= MANY
- person ::= FIRST
- person ::= SECOND
- person ::= THIRD

The right-hand sides are all non-terminals, and assuming that these non-terminals do not appear on the left-hand side of any rule, they cannot be accidentally generated from the corpus. The only way in which they can appear in a parser output tree structure is as a property of another non-terminal.

The properties of the left-hand side are listed as a set of equations in curly braces after the left-hand side. When the same property appears multiple times, then the property is assumed to have *all* of the specified values. For example:

- watch_verb {person = FIRST, person = SECOND, number = ONE} ::= “watch”
- watch_verb {person = THIRD, number = ONE} ::= “watches”
- watch_verb {number = MANY} ::= “watch”

Where a property does not appear at all, it is assumed to have any possible value, as is the case with the *gender* property above. Thus, an English translation of the above definitions might be: “the word ‘watches’ implies a verb of third person singular, any gender; while the word ‘watch’ may be plural, or may be first or second person singular, any gender.”

Notice that above, we used the left-hand side (non-terminal) *watch_verb* rather than *verb*. We use this arbitrary name as the canonical form. We can attach properties to the canonical form, such as *gloss* and *thematic role*:

- thematic_role ::= CONSUMED
- verb {gloss = “watch”, thematic_role = CONSUMED} ::= watch_verb

This sets properties which apply equally to every form of *watch_verb*. Here we have introduced a new concept: the value of the *gloss* property is a terminal (in quotes) instead of a non-terminal. It does not matter that the terminal appears in the text, since no rules link it to the definition of gloss. This is licensed because no rules have *gloss* as their non-terminal, so *gloss* does not have a restricted set of values.

Next, we introduce a syntax which allows the symbols on the right-hand side of the rule to specify properties that must agree in order for a non-terminal to be inserted at that point. These are called *conditions* on properties. We use the same curly brace notation, since the meaning is equivalent, that a match here implies and requires that certain properties have certain values. For example:

- core ::= argument {number = ONE} nucleus {number = ONE}

With a few additional rule, we can test this rule set on real sentences:

- noun {person = THIRD, number = ONE, gender = M} ::= john_noun
- john_noun ::= “John”
- nucleus {person = THIRD, number = ONE} ::= verb {person = THIRD, number = ONE}
- argument {person = THIRD, number = ONE} ::= noun {person = THIRD, number = ONE}

Now we can see that if the parser correctly enforces agreement, then “John watches” will be parsed as a valid grammatical *core*, with the structure:

- (core (argument (noun (“John”)) nucleus (verb (“watches”))))

but “*John watch” will not.

We wish to avoid the need to define such a large number of rules which match every possible value of every property for agreement purposes, for example:

- core ::= nucleus {person = FIRST, number = ONE} argument {person = FIRST, number = ONE}
- core ::= nucleus {person = SECOND, number = ONE} argument {person = SECOND, number = ONE}

- core ::= nucleus {person = THIRD, number = ONE} argument {person = THIRD, number = ONE}
- core ::= nucleus {person = FIRST, number = MANY} argument {person = FIRST, number = MANY}
- core ::= nucleus {person = SECOND, number = MANY} argument {person = SECOND, number = MANY}
- core ::= nucleus {person = THIRD, number = MANY} argument {person = THIRD, number = MANY}

which would need to be repeated for every possible *core* structure, along with equivalents for *nucleus* and *argument*, in order to prevent the parser from outputting ungrammatical cores. We can do this by introducing the mathematical concepts of *variables* and *unification*. Variables have a value which is unique and well defined in an application of a particular rule. A rule which places conditions on the symbols on its right-hand side, where the values of those conditions are variables, is called a *parametric* rule, and the variables which are so conditioned are *parametric variables* or simply *parameters*. Such a rule can be considered as a rule multiplied by all possible, consistent values of those variables.

For example, when attempting to apply the *core* rule to a sequence of words such as "John watches", the variable *subject_number* may have the value ONE or MANY. It must not have conflicting values assigned in the same rule. Its value is assigned by *unification* when it is the value of a property on the right-hand side of a rule. The process of unification merely ensures that whatever value is assigned to the variable, all assignments agree on that value. Variable names are enclosed in angle brackets. For example:

- core ::= nucleus {person = <subject-person>, number = <subject-number>} argument {person = <subject-person>, number = <subject-number>}

This rule states that within the specified core structure, the verb and argument must agree in person and number. In most languages, they would also be required to agree in gender, which can be expressed in the same way.

In the rule above, the values of the variables *subject_person* and *subject_number* are not visible outside of the rule. For example, they do not appear in the properties of the core object. It may be desirable to make them appear, for example to allow searching for all first person singular cores in the corpus. This can be done by another small extension to the syntax, which allows variables to appear as the values of properties on the left-hand side non-terminal, thus:

- core {person = <subject-person>, number = <subject-number>} ::= nucleus {person = <subject-person>, number = <subject-number>} argument {person = <subject-person>, number = <subject-number>}
- nucleus {person = <subject-person>, number = <subject-number>} ::= verb {person = <subject-person>, number = <subject-number>}
- aurgument {person = <subject-person>, number = <subject-number>} ::= noun {person = <subject-person>, number = <subject-number>}

The meaning of the variables on the left-hand side is slightly different, but only slightly. Whereas on the right hand side, all values of a variable agree, and this is enforced by unification, on the left-hand side the agreeing value is simply assigned to the property, since there is nothing (yet) to agree with, because rules are applied in a bottom-up fashion. Rules which are later invoked, which involve a *core* as a non-terminal on the right-hand side, may of course place conditions on the person and subject of that core.

It may be necessary to clarify the process of unification slightly. If none of the symbols on the right-hand side of a *parametric* rule define values for a *parameter*, then the variable has all possible values. In this case, any assignments of the variable to a property on the left-hand side would have

all possible values as well. Rather than hypothesising a rule instance for each possible value, which could quickly get out of hand, we simply omit the property entirely from the resulting instantiation of the rule.

Using the example above, if we were to introduce gender agreement as a condition, and make the resulting agreed gender a property of the core, and neither “John” nor “watches” defined a gender, then we could hypothesise a core whose gender is unspecified.

If the unification process finds that multiple possible values of the variables will satisfy the conditions, then we could hypothesise an instance of the core where the property has all the allowed values, but none of the disallowed values. For example, if “John” could be masculine or feminine but not neuter, then the core would have properties such as {gender = M, gender = F}.

Finally, if the unification process finds multiple possible sets of values for its variables, which are internally consistent but conflict with each other, then we have no choice but to hypothesise two different *core* instances. For example, if “John” could be masculine singular third person or feminine plural first person, and “watches” was compatible with both, then we would create two alternative hypothetical cores, one for each possibility.

We have described a method of storing both dictionary entries (terminals) and parser rules (non-terminals) in a single lexicon database, where each entry has the following properties:

- a non-terminal (left-hand side); and
- a list of properties with values; and
- a permutability flag, which applies to the entire right-hand side; and
- an ordered list of symbols (right-hand side).

Each symbol on the right-hand side has:

- a specification of whether it is optional or repeatable; and
- a list of conditions which must apply to its properties, with values.

The conditions and values of properties must be either:

- a terminal (literal string); or
- a non-terminal which has one or more possible values specified as rules in the rule set; or
- a variable whose value is determined by unification over all symbols in the rule.

The user interface of the software package must allow the entries in the lexicon to be edited easily. It must also be able to demonstrate to users the cascading effects of proposed changes, that is, a list of places in the corpus where the number or structure of automatically parsed structures would be changed as a result of the proposed change to the rules.

In addition, we may wish to specifically flag those clauses or structures in the corpus where:

- an automatic parsing has been selected by the user as the correct parsing, would no longer generate that same parsing under the new rules; or
- a parsing has been manually entered, overriding the automatic parsings, and an automatic parsing under the new rules now matches the one which was manually entered.

According to RRG theory, not every rule-based parsing corresponds to a valid syntactic template in the language. This condition is necessary because parser rules tend to over-generate and produce multiple possible (ambiguous) parsings. Therefore, it is necessary to be able to store template trees, for example of the form (clause (core (nucleus (predicate) argument (XP)))). The contents of each level of depth in the template can be specified by restricting the rules that can apply. This can be done by introducing a new property, *template*, which forces only certain rules to be allowed to apply within other rules.