Tools for Grammar Engineering

Gregor Erbach
University of the Saarland
Computational Linguistics
W-6600 Saarbrücken
Federal Republic of Germany
erbach@coli.uni-sb.de

We describe a tool for the development and verification of broad-coverage grammars that are to be used for both analysis and generation. Such a tool is used to ensure that the coverage of the grammar is sufficient (in logical terms the completeness of the grammar) and to control over-generation (the correctness of the grammar).

Introduction

For practically applied natural language processing systems, grammars with extensive coverage are required. The writing of broad-coverage grammars is so complex a task that it cannot be done on paper alone, but must be supported by powerful tools for testing the grammar with respect to consistency, coverage, overgeneration and accuracy.

Grammar writing is similar to programming in that grammars and programs must be tested and debugged until their input/output behaviour meets the given specifications and they run efficiently. Unlike programming, which can be approached by techniques like top-down refinement, modularization and so on, grammar writing is an incremental process, which consists of a cycle of

- writing or modifying of the grammar,
- testing of the grammar,
- debugging the grammar.

Grammar engineering tools must support this work cycle.

All existing grammar engineering tools include an editor for modification of the grammar, possibly enhanced by syntax checking, a locate function for quick access to rules and lexical entries. More advanced systems provide structure-oriented editing or a graphical editor for feature structures.

Most grammar engineering tools are built around a (chart) parser, which is used to test the grammar by parsing sentences. The parse results can be visualized and inspected in detail. Generally, the chart, phrase structure trees and feature structures can be displayed graphically.

This work was supported by IBM Germany's LILOG project. I would like to thank Roman Georg Arens, Jochen Dörre, Tibor Kiss, Ingo Raasch, Hans Uszkoreit and Jan Wilms for useful discussion.

These modes of presentation are often linked such that it is possible to select an edge of the chart or a node of the tree, and view the corresponding (sub)tree or feature structure or definition in the source file.

Few systems give diagnostic output that shows where a unification failure has occurred.

While a tool built around a parser is useful for checking the coverage and input/output behavior of the grammar, it does not help to control overgeneration, and is not very useful in locating errors in the grammar which are due to unification failure.

We propose a grammar engineering tool consisting of a parser for checking coverage, a generator for controlling overgeneration, debugging and documentation tools.

Tools for checking coverage and efficiency

For checking the coverage of the grammar, we use a bottom-up chart parser for the grammar formalism STUF (Dörre 1991). The parser is designed to support and encourage experimentation with different grammars and processing strategies (Erbach 1991).

In addition to charts, trees, and feature structures, the parser provides extensive statistics about the parsing process:

- the time needed for finding the first analysis and for finding all analyses,
- the number of possible parsing tasks,
- the number of parsing tasks on the agenda,
- the number of successful parsing tasks (chart items),
- the number of chart items that are used in a result tree.

The last three statistic data are available for each grammar rule. In this way it is possible to define parsing strategies that are sensitive to the rule which is involved in the parsing task. A good parsing strategy will delay rules that are often unsuccessful.

The tool includes a test suite, that it a set of sentences covering various syntactic phenomena, and also a set of ill-formed strings. Semantic representations are associated with the well-formed sentences.

Testing of the grammar involves parsing the strings in the test suite and checking whether the parse results contain the

correct semantic representation(s), and whether the ill-formed strings are correctly rejected.

The converse form of testing, giving semantic representations to a semantic generator and checking whether it produces the correct strings is necessary for evaluating the semantic coverage of the grammar.

Tools for controlling overgeneration

Our tool includes a generator whose task it is to generate a representative set of sentences in order to check whether the grammar overgenerates.

Before turning to the question what constitutes a representative sample of a language, we describe the algorithm used for generation [Erbach and Arens 1991].

The algorithm builds up successively longer constituents from shorter ones. The initial constituents of length 1 are the words from the lexicon. More constituents of length 1 are added by applying unary rules to the constituents of length 1. Constituents of length n are built by applying a binary rule to constituents of length x and length y, such that x+y=n, or by applying a unary rule to a constituent of length n.

Since in general, languages are infinite, it is not possible to generate all sentences. The language generated can be limited by

- setting a maximal sentence lenth,
- limiting the initial lexicon, so that only one member of a class of lexical entries is present,
- excluding certain rules of the grammar, so that certain constructions that one is presently not interested in are avoided, e.g. relative clauses,
- limiting recursion,
- filtering the constituents generated according to the grammar writer's interests, for example only sentences and noun phrases.

All of these devices which limit the grammar must be used with caution, in order to avoid losing interesting examples in the language sample generated.

When looking through the generated language sample, the user may select any sentence, and view its feature structure and derivation tree, using the tools provided with the parser.

Debugging tools

Sometimes, it is desirable to know how certain linguistic descriptions fit together. For example, when debugging a grammar, one might want to see what happens if a particular rule is applied to two particular lexical items, or whether or not two feature structures unify. With a parser-based tool, one must make up a sentence in which such a configuration occurs, and then inspect the parse result.

We provide this functionality more directly. The user may select any linguistic object (lexical entry, type definition or chart item) and unify it with any other linguistic object. If grammar rules are seen as local trees, the user may also unify any linguistic unit with any of the nodes in the local tree defined by the grammar rule.

While unification failures are not kept track of during parsing for efficiency reasons, this structure builder will show exactly which features have caused a unification to fail, and thus assist in the diagnosis of errors in the grammar.

Documentation tools

Good documentation of a grammar is a prerequisite for modifying or extending it or re-using it for another application, and should be supported and by appropriate tools. We believe that documentation tools are best integrated with a grammar engineering tool. There are various reasons for doing so.

First of all, the tool can be constructed such that it reminds the user of including documentation in the source files and keeps track of when definitions in the grammar were changed, and by whom.

Second, integration of grammar engineering and documentation tools makes it easy to include output of the parser (charts, trees and feature strctures) into the documentation that is being written.

Third, we assume that the technical documentation is a hypertext. In addition to the standard hypertext links from one piece of text to another, we can include links from the text to the functionality of the grammar engineering tool. By selecting a word, or the name of a grammar rule or a type definition in the text of the documentation, a link will be built that allows the user to either view its definition in the source file or display its feature structure. By selecting an example sentence in the documentation text, it is possible to activate the parser to view its chart, tree or feature structure.

Implementation

The tool described here has been partially implemented in Quintus Prolog and X-Windows under AIX on an IBM PS/2. It is integrated into the text understanding system LEU/2 developed in IBM Germany's LILOG project.

The parser was implemented by the author, the grammar formalism and the user interface by Jochen Dörre and Ingo Raasch and the generator by the author and Roman G. Arens.

References

Gregor Erbach and Roman Georg Arens. Evaluation von Grammatiken für die Analyse natürlicher Sprache durch Generierung einer repräsentativen Satzmenge. In *Proceedings of GWAI -91,* pages 126-129, Bonn, September 1991, Springer.

Jochen Dörre. The Language of STUF. In: O. Herzog and C.-R. Rollinger (eds.): *Text Understanding in LILOG*, pages 33-38, Springer, Berlin, 1991.

Gregor Erbach. A Flexible Parser for a Linguistic Development Environment. In: O. Herzog and C.-R. Rollinger (eds.): *Text Understanding in LILOG*, pages 74-87, Springer, Berlin, 1991.