A PARSING METHODOLOGY FOR ERROR DETECTION

Davide Turcato[§] **Devlan Nicholson**[§] **Trude Heift**[†]
Janine Toole[§] **Stavroula Tsiplakou**[†]
Simon Fraser University, Burnaby Mountain, BC, Canada V5A 1S6
[§]Natural Language Lab, School of Computing Science
{turk,devlan,toole}@cs.sfu.ca
[†] Department of Linguistics
{heift,stavroula_tsiplakou}@sfu.ca

1 Introduction

We discuss the design of grammars for syntactic error detection. The topic is equally relevant for grammar checkers and Intelligent Language Tutoring Systems (henceforth ILTS). The proposed methodology addresses three interrelated issues recurrent in the error detection literature:

Efficiency. In error detection the search space is larger than in ordinary parsing. Therefore there is a need to keep the search space manageable and to introduce some control mechanism over parsing.

Modularity. Different knowledge sources need to be used, some of which are domain-dependent. In order to combine efficiency with modularity, different knowledge sources should be amenable of being stored separately but accessed in parallel at parsing time. A related issue is also the ability to reuse the same grammar for both error detection and ordinary parsing.

Expressive power. There is a tendency to use unification-based frameworks of the same sort used for ordinary parsing. However, a more powerful machinery is needed for error detection than for ordinary parsing.

Although the different requirements of error detection have been separately addressed in various ways, their contemporaneous satisfaction can be problematic, due to the drawbacks that solutions to one requirements can imply for other requirements. In the following, we show how an extended unification-based formalism can be used to satisfactorily address all the different requirements listed above at the same time. The described ideas have been implemented in a grammar for error detection, embedded in an ILTS for Modern Greek.

2 Methodology description

The task of our ILTS grammar module is to analyze an input sentence and return a list of violations, along with a list of 'non-violations', i.e. correct instances of grammatical phenomena occurring in a sentence, used to update an individual student model.

Expressive power. We take an approach to error detection based on a unification grammar extended via procedural attachments. We defined a typed feature structure formalism with definite clause logic programming attachments, akin to existing formalisms like ALE [1]. In such formalisms, not only unification, but also any other operation that can be defined in terms of the underlying logic programming language can be performed on linguistic descriptions. The advantage of this approach is twofold: (i) procedures of any complexity can be encoded, while keeping a single control structure driven by a declarative grammar; (ii) any number of additional knowledge sources can be separately encoded, while being all accessible at parsing time.

Modularity. One of our design guideline was to develop a *multi-functional* grammar: we wanted to be able to use the same grammar for both ordinary parsing and error detection. This can be achieved by defining attachments in a consistent way. Whenever a feature is relevant to error detection, any check on its value is done in some suitable attachment to the relevant rule, instead of the rule body. Moreover, each predicate is systematically defined as a disjunction: one clause performs whatever checks and actions are needed for error detection, the other one performs whatever standard unification is needed for ordinary parsing. The two clauses of each predicate are mutually exclusive: the former is only used when error detection is needed, the latter when only ordinary parsing is. To this end we define environment variables, again in the form of definite clause, which we use as system parameters.

All violation-related attachments are uniformly handled in three stages: (i) A *test* on the relevant features, where violations are detected; (ii) A *selection* of the action to be taken, where detected violation are interpreted, in terms of some grammatical configuration; (iii) The *execution* of the selected action, where a grammatical configuration is mapped onto an appropriate label to be added to the list of violation or non-violation, respectively. This processing scheme allows one to decouple constraint relaxation from error (or non-error) flagging, which is based on a number of other factors, independently handled. If the system were ported to a different domain, the use of a different inventory of violations in the error flagging phase would not necessarily involve changes in the error detection phase. Different error inventories can be alternatively used by setting an appropriate system parameter.

Efficiency. We implemented a mechanism that allows user-defined control strategies on parsing. A control strategy is defined via a definite clause, which takes as arguments a *linguistic description* and a *priority value*. Given a linguistic description, the clause assigns a numeric value, representing the description's priority in the parsing agenda. Edges in the parsing chart are retrieved from the agenda and asserted according to the priority value associated to them by the priority clause. In this way, parses are obtained in the order defined by a user, according to some relevant criterion. Different parsing strategies can be defined and the system can be switched between them, again by means of system parameters. For example, we use different parsing strategies depending on whether error detection or ordinary parsing is performed.

In the proposed architecture, a single control structure represented by a declarative unification grammar can be integrated with any number of further knowledge sources and procedures of any complexity, and provided with means to define a parsing strategy based on any information available. The result is a modular integrated system whose behavior can be changed between different modes by simply setting some general system parameters.

References

[1] Bob Carpenter and Gerald Penn. ALE. the Attribute Logic Engine user's guide. version 3.2 beta. Technical report, Bell Laboratories and Universität Tübingen, Murray Hill, NJ, USA and Tübingen, Germany, May 1999.