PARSING IN DIALOGUE SYSTEMS USING TYPED FEATURE STRUCTURES

Rieks op den Akker, Hugo ter Doest, Mark Moll and Anton Nijholt
Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede
e-mail: {infrieks,terdoest,moll,anijholt}@cs.utwente.nl

1 Abstract

The analysis of natural language in the context of keyboard-driven dialogue systems is the central issue addressed in this paper. A module that corrects typing errors, performs domain-specific morphological analysis is developed. A parser for typed unification grammars is designed and implemented in C++; for description of the lexicon and the grammer a specialised specification language is developed. It is argued that typed unification grammars and especially the newly developed specification language are convenient formalisms for describing natural language use in dialogue systems. Research on these issues is carried out in the context of the SCHISMA project, a research project in linguistic engineering; participants in SCHISMA are KPN Research and the University of Twente.

2 The Preprocessor MAF

As we postponed the development of a spoken interface to the SCHISMA system, we concentrate here on the analysis of keyboard input. Thus the input of the MAF module is the character string typed in by the client. The MAF module is best seen as the preprocessor of the SCHISMA system. It handles typing errors and detects certain types of phrases (proper names that occur in the database, date and time phrases, number names, etc.). The latter task of MAF is especially important, since it extracts information crucial for the continuation of the dialogue from the input string.

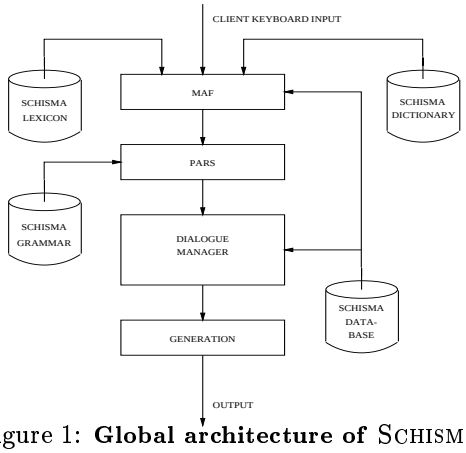

Figure 1: **Global architecture of** SCHISMA

Output of the MAF module is a *word graph*. We define a word graph here as a directed graph having as its nodes the positions in the input string identified as (possible) word boundaries. Nodes are numbered starting with 0 for the leftmost boundary; that is the position left to the first input character. A pair $(index_1, index_2)$ is an edge of the graph if $index_1$ and $index_2$ are word boundaries, $index_1 < index_2$ and the words enclosed between $index_1$ and $index_2$ are identified as one text unit; that is one or more words are identified by MAF as a lexical item to be provided to the parser as a whole. In addition, the MAF module labels the edges of the graph with a value m that indicates the quality of the recognition (and maybe correction) performed.

On the implementation level this means that the MAF module has as output a collection of items (rd, m) where rd is a 3-tuple $(fstruct, index_1, index_2)$, $fstruct$ a typed feature structure, $index_1$ and $index_2$ indices on the word level as explained above, and m is a value indicating the plausibility of rd as a representation of (part of) the input string.

The architecture of the MAF module is quite simple: an error correcting module accepts the input string, processes it, sends its output to some tagging modules and these send their output to the module for morphological analysis and lexicon lookup.

The error correcting module ERROR outputs a word graph that is provided to the tagging modules PROPER, NUMBER, DATE and

TIME that scan the graph for phrases that have special meaning in the SCHISMA domain. In addition, the word graph is provided to the MORPH/LEX module. For performing the error correction ERROR has access to a large dictionary (typically 200,000 words). The tagging modules look for phrases in the input string that carry particularly important information for the dialogue; especially the detection of proper names referring to database items, phrases indicating date and time information and number names is aimed at here; for detecting proper names referring to the database the PROPER module needs access to the SCHISMA database. The output of the taggers then is provided to the MORPH/LEX module; MORPH/LEX creates items for the parser out of the tag information provided by the taggers and it searches the word graph for words that appear in the domain-specific lexicon and for which domain-dependent semantic information is recorded in it.

For details on the tagging modules and the phrases they recognise we refer to (Op den Akker et al. 1995). Also the ERROR and MORPH/LEX module are treated in depth there.

3 The Specification Language

To specify a language it is necessary to have a metalanguage. Almost always the usage of a specification language is limited to only one grammar formalism. This is not necessarily a drawback, as such a specification language can be better tailored towards the peculiarities of the formalism. For example, Carpenter's ALE is a very powerful (type) specification language for the domain of unification-based grammar formalisms. But apart from expressiveness of the specification language, the ease with which the intended information about a language can be encoded is also important. An example of a language that combines expressiveness with ease of use is Alshawi's Core Language Engine. Unfortunately the Core Language Engine (CLE) does not support typing. Within our project a type specification language has been developed that can be positioned somewhere between ALE and CLE. This specification language (called $\mathcal{TFS}$) can be used to specify a type lattice, a lexicon and a unification grammar for a head-corner parser. The notation is loosely based on CLE, though far less extensive. For instance, the usage of lambda calculus is not supported.

The following example shows how a type lattice can be specified.

```
TYPE(performance;entity;<constr>;<QLF>)
TYPE(play;performance;<constr>;<QLF>)
TYPE(concert;performance;<constr>;<QLF>)
TYPE(musical;play,concert;<constr>;<QLF>)
TYPE(ballet;concert;<constr>;<QLF>)
```

A type specification consists of four parts: a type id for the type to be specified, a list of supertypes, a list of constraints and a formula expressing the semantics for the new type. For each type `<constr>` should be replaced with PATR-II-like path equations and `<QLF>` should be replaced with the semantics in a quasi-logical form. The idea is that the constraints are only necessary *during* parsing and the semantics are passed on to be used *after* parsing.

The next example shows how typing can make some grammar rules superfluous.

```
TYPE(perfphrase; nounphrase; ; )
RULE(nounphrase --> *perfphrase*;
  <nounphrase kind> = <perfphrase kind>,
  <nounphrase sem> = <perfphrase sem>)
```

The asterisks mark the head in the grammar rule. Both the type and rule specify that a performance phrase is a kind of noun phrase.

By path equations QLF expressions can be passed on to other constituents. In the following example a possible quasi-logical form for a phrase is given:

the opera performance on the 4th of January

```
EXISTS X (opera(X) AND date(X,4-1-95))
```

The `opera` predicate comes from the QLF part of the opera type and the `date` predicate is generated by parsing the time phrase. Another grammar rule combines these predicates and binds the variable `X`.

It also possible to bind unbound variables to a certain value. This can be done in the specification of a subtype, a word as well as a grammar rule.

References

Op den Akker, R., Ter Doest, H., Moll, M., and Nijholt, A. (1995). Parsing in dialogue systems using typed feature structures. Memoranda Informatica 95-25, Department of Computer Science, University of Twente.