Using lexical functional grammars for NL systems

Javier Blanco and Pablo Duboué

Facultad de Matemática, Astronomía y Física (FAMAF)
Ciudad Universitaria, 5000, Córdoba, República Argentina.
E-mail: blanco@mate.uncor.edu
E-mail: pablod@hal.famaf.unc.edu.ar

Abstract. The theory of Lexical Functional Grammars (LFG) was created at the beginning of the 1980's and it is a linguistic formalism based on a descriptive, unification-based theory. This formalism has been usually tackled with the use of unification based approach (logic programming). In this work we describe a Haskell parser for LFG, parametric with respect to the grammar and the lexicon, targeting the resolution of the symbolic equations' system generated by the theory.

1 Introduction

This article describes a functional parser for an specific linguistic theory. The necessary linguistic background will be introduced in section 1.1, the proposed parser will be analyzed in section 2. In section 3 we will describe the unification algorithm that is the core of the parser. Due to space restrictions only a high level analysis of the algorithm will be presented. For a full description, you may refer to [Dub98] (copies are available (electronically) from the authors). Finally, some results and further extensions will be discussed in the conclusions.

1.1 LFG

The theory of Lexical Functional Grammars was introduced by Joan Bresnan and Ronald Kaplan [BK82]. It is considered one of the main modern syntactic theories in the field of linguistics. A lexical functional grammar assigns two levels of syntactic description to every sentence of a language: a *constituent structure* and a *functional structure*. Constituent structures (c-structures) characterize the phrase structure configurations as a conventional phrase structure tree. Surface grammatical functions such as *subject*, *object* and *adjuncts* are represented in the functional structure (f-structure).

The c-structure is the parse tree, or set of parse trees (in the presence of an ambiguous grammar), generated by a set of context-free rules. This context free rules are extended by the use of functional schemata, which are a sets of equations that decorate each symbol. The format of an LFG rule is shown in figure 1.

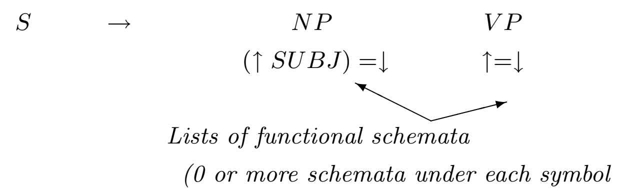
$S \quad \rightarrow \quad NP \quad VP$
$(\uparrow SUBJ) = \downarrow \quad \uparrow = \downarrow$
Lists of functional schemata
(0 or more schemata under each symbol

Fig. 1. Format of LFG rules.

These functional annotations are instantiated to provide a formal description of the f-structure. The smallest structure satisfying those constraints is the grammatically f-structure associated with the sentence. An f-structure is a hierarchical attribute-value matrix. We will introduce f-structures by means of an example (for a full definition see [Kap89]). An f-structure could be seen on figure 2. It can be read as follows: the f-structure f_5 contains a line with $SUBJ$ as attribute and the f-structure f_2 as value. In symbols, this is written as $(f_5\ SUBJ) = f_2$. Accordingly, we can read in the figure that the f-structure f_3 has a line with attribute NUM and value 3. In other words, $(f_3\ NUM) = 3$. Since the equation $f_3 = f_2$ holds, we can deduce $(f_5\ SUBJ\ NUM) = f_3$. The idea behind LFG is to relate the functional annotations in the rules with a formal system of equations of the kind we have just described.

$$\begin{array}{c} f_1 \\ f_4 \\ f_5 \end{array} \left[\begin{array}{ll} PRED & \text{`SEE} < (f_5\ SUBJ)(f_5\ OBJ) >\text{'} \\ SUBJ & \begin{array}{c} f_2 \\ f_3 \end{array} \left[ \begin{array}{ll} PRED & \text{`JOHN'} \\ NUM & SING \\ PERS & 3 \end{array} \right] \\ OBJ & \begin{array}{c} f_6 \\ f_7 \end{array} \left[\begin{array}{ll} PRED & \text{`MARY'} \\ NUM & SING \\ PERS & 3 \end{array} \right] \end{array} \right]$$

Fig. 2. An f-structure.

The theory also got part of its name from the important role played by the lexicon. Several issues —such as some kinds of agreement— are resolved at the lexical level. The high complexity of the lexicon allows to cope with some of the ambiguities of the natural languages (e.g. the same word belonging to many grammatical classes). The lexical entries are depicted in figure 3.

The complete version of this work includes a deeper understanding of the theory by means of a fully-detailed example.

2 Our work

NLC

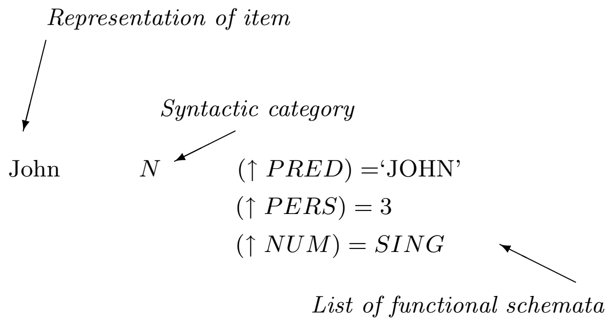

Fig. 3. Format of LFG lexical items.

Following the lines of the previous sections we built a parser for LFG written in the Haskell functional language. Our approach consists of an interpreter, i.e., we take the grammar and lexicon as parameters, together with the sentence to be parsed. This approach, although less efficent, encourages experimentation over different lexicons and grammars.

2.1 Main function

The main function receives as parameters data structures representing the grammar (`GramF`), and the lexicon (`LexiconF`), and also the `String` to be parsed, returning an f-structure (`FEstr`). Since it is possible for the parser to find that the sentence is not syntactically correct, we follow normal practice in functional programming by representing this kind of failures by an empty list [Wad85]. This may be used also to produce more than a parse tree in case of ambiguous grammars. Nevertheless, the solver will always work with just one parse tree, returning at most one f-structure. This fact provides a good separation of concerns. Therefore, the function `lfg` has the following type:

```
lfg :: GramF -> LexiconF -> String -> [FEstr]
```

The `lfg` function is defined by a serie of sub functions that perform one step forward toward the completion of the f-structure. The figure 4 shows the sub function division.

`parser`. This function performs the generation of the parse tree that is produced by the context free grammar. To accomplish his task it uses the monadic parser combinators, introduced by [Hut92], [HM96] and [Wad95]. In this work we do not define a lexer. One may think of this absence as a flaw, because this solution scales up unsatisfactorily. However, for small lexicons it works fine and it is a nice example of the expressive power of the parser combinators. Special care has been taken during the construction of the tree to preserve the functional annotations. That will decorate each node in the final tree (`FTree`).

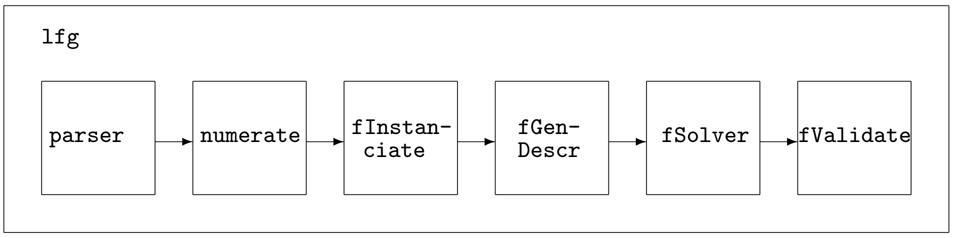

Fig. 4. Main function description (`lfg` function).

`numerate`. A function that makes use of a state monad [Wad92] and traverse the whole `FTree`, assigning a number to each node.

`fInstanciate`. As described in the introduction, this function resolves the meta variables $MOTHER$ (↑) and $SELF$ (↓) by using the numbers generated by `numerate`.

`fGenDescr`. After the instantiation we are ready to forget about the tree and keep only the equations. This function accomplishes this task by traversing the tree and picking up the equations at each node.

`fSolver`. By using monadic parser combinators, the most difficult points of `lfg` have moved from `parser` to this function. In order to solve the system of simultaneous symbolic equations, we use a tailored unification algorithm as outlined in [Wes89]. This algorithm is equivalent to the unification algorithm for feature structures, represented as direct acyclic graphs (DAGs), as described in [All95]. This function is the topic of next section.

`fValidate`. The completeness and correctness checks we mentioned in the introduction are performed by this function. It behaves as an `FEstr` filter.

2.2 Input files

The `GramF` and `LexiconF` data structures are provided from text files. The format of the text files is shown in the figure 5 using EBNF format. Here we have glued together both EBNF texts, the `Gram` non-terminal generates the grammar file format, whereas the `Lex` generates the lexicon file format. An example is shown in figure 6.

2.3 Interface

In order to take full advantage of the parametric design of the parser, we need a better interface than the raw command-line. Following [MvD96] and using available Haskell libraries for writting CGI application we developped a fully functional front-end for the LFG engine. Its main features include:

```
Gram       → (FRule ';')*
FRule      → Ident '->' SintFSch*
Ident      → alphanum+
SintFSch   → Ident '(' (FSch ';')* ')'
FSch       → FExpr '=' FExpr
FExpr      → FEstrExpr | FValue
FEstrExpr  → Mother | Self | Apl
Mother     → 'up'
Self       → 'dn'
Apl        → '(' FEstrExpr Ident ')'
FValue     → Pred | DeInt | DeSint
Pred       → ''' Ident '<' Ident* '>' '''
DeInt      → integer
DeSint     → Ident
Lex        → (LEntry ';')*
LEntry     → Ident Ident '(' (FSch ',')* ')'
```

Fig. 5. EBNF description of the input files.

Grammar file

```
np -> n  (up = dn);
s  -> np ((up subj) = dn) vp (up = dn);
vp -> v  (up = dn) np ((up obj) = dn)
```

Lexicon file

```
John n (pred='JOHN <>', pers=3, num=SING, gen=M);
Mary n (pred='MARY <>', num=SING, gen=F);
sees v (pred='SEE <SUBJ OBJ>', time=PRES, pers=3)
```

Fig. 6. Input files example.

multi-user. Assuming that if plugged to the Internet this kind of tool could be useful for different people, we decided to build the front-end around a concept of accounts protected with passwords. Each account has its own set of (private) grammar and lexicon files.

fully CGI compliant. By this term we mean the fact that our application make no use of fancy technologies such as *Java*^{TM}, *JavaScript* or even *cookies*. This increases the usefullness of the front-end, since it is practical even under the `Lynx` web browser (a text-mode only browser).

high performance. Our first approach was to use Hugs 1.4 (*Haskell User's Gofer System*) through its scripting interface, `runhugs`, and continue using a Haskell interpreter. However, the lack of file functions and some performance problems compeled us to a compiled solution. We ended up using the *Glasgow Haskell Compiler*, `ghc` version 2.10 under **Linux**.

Due to restrictions of space, we do not include any more details about the interface. The interested reader may find them in [Dub98]. The interface can be viewed on-line in

$$\texttt{http}://\texttt{hal.famaf.unc.edu.ar}/\sim\texttt{pablod/lfg}/$$

.

3 The unifier

As was previously stated, this algorithm follows [Wes89]. It can be described as follows:

1. All the equations are rewritten to some of the following canonical forms:

$$f_i = f_j \ (f_i\ A) = B \ (f_i\ A) = f_j \tag{1}$$

   Note that any other equations, either of the form $((f_i\ A)B) = C$ or $((f_i\ A)B) = f_j$, can be translated to this form by the introduction of auxiliary variables.
2. Among the equations in canonical form, we distinguish two types: (Type 1) $(f_i\ A) = B$ and $(f_i\ A) = f_j$ and (Type 2) $f_i = f_j$.
3. A working data structure `FTable` is built (see 3.1 for an explanation of the `FTable` data structure).
4. Using the FTable the information of the equations of Type 1 is introduced in the table.
5. Now the equations of Type 2 are taken into account. Here we produce the **merge** (the merging process is a kind of unification process, it is explained in 3.2) of the two f-structures as the result of the operation. If no merging is possible, we are in front of a non-grammatical sentence.

3.1 The table

This data structure is the functional language equivalent to an array of pointers to f-structures. The idea is to have all the f-structures indexed by their names (and *alias*), up to one level. That means that if one has an f-structure f_5 which has a line with key $SUBJ$ and has as value another f-structure, say f_9, then in the `FTable` you will have an entry for f_5, another entry for f_9 and a $SUBJ$'s line in the f_5's entry containing a **pointer** to f_9. A graphical representation of the `FTable` can be seen on figure 7.

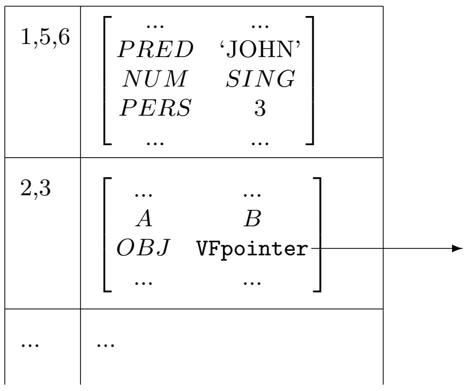

Fig. 7. `FTable`.

3.2 Merging

The merging process is a kind of unification that takes two variables as parameters and unifies them according to the following algorithm:

The merge of two instances of the same atomic name consists of that atomic name. Atomic names which are not identical do not merge. Semantic forms (flanked by '...') never merge. To achieve the merge of two f-structures —let us call them f_m and f_n— we select one of these f-structures, say f_m, and for each attribute a in f_m, we attempt to find an instance of a in f_n. Let us call the value associated with a in f_m v. If a does not occur in f_n, then we add the attribute a and the value v to f_n. Contrarily, if a is already present in f_n, and its value is v', then the merge of v and v' becomes the new value of a in f_n. If all of the subsidiary mergers are successful, then the modified version of f_n represents the merge of f_m and f_n.

3.3 Pseudo-code

The algorithm can be sketched as the following functional pseudo-code:

```
fSolver :: FDescr -> FTable
fSolver descr =
    (fSolver2 descr1) .
    (fSolver1 descr1) emptyTable                                  (2)
  where
    descr' = cleanUpDescr descr
    (descr1, descr2) = splitEquations descr'
```

The solver for the Type 1 equations:

```
fSolver1 :: FDescr -> FTable -> FTable
fSolver1 [] table = table
fSolver1 ((EQUALS e1 e2):rest) table
  | (isOfTheFormF_A e1) & & (isOfTheFormV e2) =
    let (f,a) = getF_A e1
        v   = getV e2 in                                          (3)
    (fSolver1 rest) . (add f (a,v)) table
  | (isOfTheFormF_A e1) && (isOfTheFormF e2) =
    let (f,a) = getF_A e1
        f'  = getF e2 in
    (fSolver1 rest) . (add f (a,f')) table
```

The solver for the Type 2 equations:

```
fSolver2 :: FDescr -> FTable -> FTable
fSolver2 [] table = table
fSolver2 ((EQUALS e1 e2):rest) table
  let f1 = getF e1                                                (4)
      f2 = getF e2 in
    (fSolver2 rest) . (merge f1 f2) table
```

The previous codes are oversimplified in several ways: for example, all the exception handling is done using the Maybe monad included in Haskell [PH+96]. The code for the merge and add functions is not included because it is too dependent of the data-structures we have used to represent the different data types.

In order to simplify our description we have included an informal description of both functions:

`add` This function takes an f-structure pointer, a new line and a table and returns the table which results from adding that line to the f-structure referred by the pointer. Special care should be taken regarding the creation of new f-structures (if the pointer is not defined in the table). The consistency of the table is maintained by making a merge if the line was already present.

`merge` The previously discussed merging algorithm is implemented by this function. It takes the pointer of the two f-structures supposed to get merged together, the working table and gets back the resulting table (if any).

4 Conclusions

This tool was used to generate an Spanish LFG grammar which capturated several characteristics of this language. In particular, it managed well to parse adjective relative clauses, to support subjet-verb agreement in person and number, to support adjective-noun agreement both in gender and number, to enforce verb subcategorization and to mark absent subjects (a feature not present in English but quite usual in Spanish). We had troubles with the fact that the lexer works in a word per word fashion not allowing us to define groups of words that plays the role of a single word, for example, an adverbial phrase. Some restrictions imposed upon the syntax of the equations showed too restrictive (there was no straightforward construction to capture common Spanish adjunct constructions). For further details see the discussion in [Dub98].

4.1 Further work

This work can be extended in several ways. Besides extending the subset of the LFG theory supported, the system could be targeted toward real use and large scale dictionaries. We feel quite confident about the scalability of a functional language solution (contrasting with logical programming based approaches). However, to accomplish this goal, the algorithms should be dramatically improved. Following [Lee93] the use of monadic parser combinators could be maintained by changing to a compiled view (the gammar and the lexicon would be no longer parameters) and using a functional language with internal use of **memoizing functions**. When large databases are used, a lexer is required. A technique for lexer construction that suits well our work is finite state transducers as described in [Ant90]. The unification algorithm is where our efforts are still targeted. It seems that it can be improved and clarified in several ways.

In its present state, we have presented a tool that could be interesting for research in linguistics. The current thread of research in LFG is extending the expressive power of the theory. Therefore, it is our intention to extend the system to cope with feature attributes as depicted in [Sad96], or at least constraining equations as described in [Kap89].

Besides its extension to cope with large corpora, this parser can be the first step in a wide range of applications involving natural language and functional programming. Some of our ideas included using Haskell functions for the semantic forms ($PRED$) and try to obtain *executable* f-structures.

To conclude, the use of lazy functional language suited perfectly the construction of the tool.

References

[All95] James Allen. *Natural language understanding 2nd ed.* The Benjamin/Cummings Publishing Company, Inc., 1995.

[Ant90] E. Antworth. Pc-kimmo: A two-level processor for morphological analysis. Technical report, Academic Computing Department, Summer Institute of Linguistics, Dallas, 1990.

[BK82] Joan Bresnan and Ronald M. Kaplan. Introduction: grammars as mental representations of language. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages xvii–lii. The MIT Press, Cambridge, MA, 1982.

[Dub98] Pablo Ariel Duboué. Desarrollo de un parser funcional para el lenguaje castellano. Technical report, FaMAF, Universidad Nacional de Córdoba, Argentina, August 1998. an electronic version is available from the author (in Spanish) e-mail `pablod@hal.famaf.unc.edu.ar`.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, December 1996.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[Kap89] Ronald M. Kaplan. The formal architecture of Lexical-Functional Grammar. In Chu-Ren Huang and Keh-Jiann Chen, editors, *Proceedings of the Republic of China Computational Linguistics Conference (ROCLING II)*, pages 3–18, Taipei, 1989. Academia Sinica. Reprinted in Mary Dalrymple, Ronald M. Kaplan, John Maxwell, and Annie Zaenen, eds., *Formal Issues in Lexical-Functional Grammar*, 7–27. Stanford: Center for the Study of Language and Information. 1995.

[Lee93] René Leermakers. *The Functional Treatment of Parsing.* Kluwer Academic Publishers, 1993.

[MvD96] Erik Meijer and Joost van Dijk. Perl for swine: Cgi programming in haskell. In *First Workshop on Functional Programming*, Buenos Aires, Argentina, 1996. URL `http://www.cs.ruu.nl/∼joostd/cgigofer.dvi`.

[PH$^+$96] John Peterson, Kevin Hammond, et al. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.3. Technical report, Yale University, May 1996.

[Sad96] Louisa Sadler. New developments in LFG. In Keith Brown and Jim Miller, editors, *Concise Encyclopedia of Syntactic Theories.* Elsevier Science, Oxford, 1996.

[Wad85] Philip Wadler. How to replace failure by a list of successes. In *2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Springer-Verlag.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. (Special issue of selected papers from 6'th Conference on Lisp and Functional Programming.).

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995. (This is a revised version of [**?**].).

[Wes89] Michael T. Wescoat. Practical instructions for working with the formalism of Lexical Functional Grammar. MS, Xerox PARC, 1989.