A Lexical Functional Grammar System in Prolog

Andreas Eisele and Jochen Dörre

Department of Linguistics
University of Stuttgart
West Germany

Abstract

This paper describes a system in PROLOG for the automatic transformation of a grammar, written in LFG formalism, into a DCG-based parser. It demonstrates the main principles of the transformation, the representation of f-structures and constraints, the treatment of long-distance dependencies, and left recursion.
Finally some problem areas of the system and possibilities for overcoming them are discussed.

Introduction

In order to improve our knowledge about natural language, it is desirable to have a high-level description language which can be used to test grammars on a computer system, but which is independent of the details of the implementation. For linguists without knowledge of programming languages, a system for writing and testing grammars on a computer should be offered.
At the University of Stuttgart such a system has been implemented in PROLOG, which uses the formalism of Lexical-Functional Grammar [Kaplan/Bresnan 82] as its description language.
The system makes it possible for the user to enter grammar rules and lexical entries directly in the form described in [Kaplan/Bresnan 82]. The input is translated into PROLOG rules, which form a top down parser in definite clause grammar style.
Equations and constraints associated with a grammar rule are evaluated as soon as the rule is used, thus allowing the rejection of incorrect parses as soon as constraints are violated.
One of the main problems using DCG grammars - the prohibition of using left-recursive grammar rules - is solved by a conversion to right-recursive rules that does not violate the semantics of the functional description.

Main Goals of the Implementation

When we started the implementation of our LFG-Sytem we had mainly the following tasks in mind:

- Independence of Implementation
The LFG system is meant to be a grammar-writer's tool which allows him to ignore completely the details of the implementation. Specifically we wanted the system to be useful for linguists without any prior knowledge of PROLOG.

- Complete Coverage of the LFG-Formalism
The system should cover all features of LFG as they are stated by [Kaplan/Bresnan 82]. This means we had to incorporate the principles of consistency, completeness and coherence, inequality, positive and negative existential constraints and long distance dependencies.

- Flexible Environment for Grammar Development
To be a really useful tool, the system must allow for testing the grammar fragment and changing it incrementally. In this point we had to find a good compromise between speed of parsing and speed of grammar modification.

- Using as much of PROLOG as possible
We wanted to profit from the facilities PROLOG offers for grammar implementors in two respects:
i) Using DCGs for parsing (and overcoming the prohibition of left recursion)
ii) Profiting from PROLOG's unification mechanism to implement LFG-Unification.

Unification

LFG is a unification-based grammar formalism. To be more precise, any defining equation in LFG can be interpreted as the unification of certain f-structures. Unifying two f-structures is an operation very similar to set union. However, unification may fail, if the stuctures contain contradicting values for the same attribute. Otherwise the two structures become the same object, which contains the information of both structures. Consider for example the LFG rule

$$\begin{array}{lll} S \rightarrow & NP & VP \\ & (\uparrow \text{SUBJ})=\downarrow & \uparrow=\downarrow \end{array}$$

and take FS, FNP and FVP as the f-structures associated to the S, NP and VP node, respectively. Then the two equations can be interpreted as the unifications

$$FS \ \underline{U} \ FVP \quad \text{and} \quad FS \ \underline{U} \ [SUBJ = FNP].$$

The unification of f-structures is also closely related to the unification of PROLOG-Terms, yet there are two important differences: In f-structures values are identified by labels (the attributes) and their number is potentially unlimited, whereas in PROLOG-terms the arguments are identified by their position and their number is fixed. In the following we show how we can model f-structure unification in PROLOG.
We represent partial f-structures as an 'open ended' list of pairs:

$$[\ A1 = V1\ ,\ A2 = V2\ ,\ \dots\ ,\ An = Vn \mid \_\ ]$$

where the Ai stand for (atomic) attributes and the Vi for the values associated to these attributes. These values are either atomic, terms denoting semantic forms, f-structures themselves or the term 'set(S)' where S stands for an open-ended list of f-structures denoting a set.
The unification of two f-structures is evaluated in this representation by inserting into both structures the features missing with respect to the other, and then PROLOG-unifying the variables that stand for the rest of the lists. The values of features which the structures have in common have to be unified recursively.
A procedure which performs this action can easily be written in PROLOG* :

```
merge(X,X) :- !.
merge([A=V1|R1],F2) :- del(A=V2,F2,R2),
                       merge(V1,V2),
                       merge(R1,R2).

del(F,[F|X],X) :- !.
del(F,[E|X],[E|Y]) :- del(F,X,Y).
```

When called with the f-structures FS1 and FS2, the predicate 'merge' recursively reduces the attributes in FS1. If an

*The treatment of sets is omitted here for the sake of simplicity.

attribute appears also in FS2, 'del' finds its value in FS2 and the two values are unified by the first recursive call of 'merge'. If an attribute is unspecified in FS2, 'del' will insert it at the end of the structure as a new attribute. Eventually, 'merge' will reach the tail variable of FS1 and instantiate it with exactly the attributes which appear in FS2 but not in FS1. After successful execution of 'merge' FS1 and FS2 contain the same attributes with the same values (maybe in different order) and tail variables at any level are shared. So any further unification affecting one of them will affect the other structure in exactly the same way.

Example: The goal

```
merge([subj  = [spec = def,
                num  = sg,
                pred = girl
                 | RSubj1]
        | R1],
      [pred  = hand(subj,obj2,obj),
       tense = present,
       subj  = [num = sg | RSubj2]
        | R2]).
```

yields the instantiations

```
R1 = [pred = hand(subj,obj2,obj), tense = present | R2]

RSubj2 = [spec = def, pred = girl | RSubj1]
```

The fact that the unification is performed by extending both of the structures and that there is no explicit result is essential when dealing with reentrant structures, i.e. embedded structures that can be reached by more than one path. In the case when such a structure is extended while it is reached by one of the possible paths, an access via a different path will also reach this extension.

Treatment of Completeness, Coherence and Constraints

In addition to defining equations, which can be mapped onto the monotonic operation of unification, LFG includes formal devices, some of which cannot be treated monotonically, but need the notion of a 'final' f-structure [Shieber 85]. More concretely, the violation of positive existential constraints, constraining equations and completeness cannot be checked before the parsing process has finished.

- *Existential constraints* are treated by inserting the attribute into the f-structure, but leaving the associated value uninstantiated (if it isn't already known). The condition that this variable must be instantiated is stored in a list (Ex_Tests) especially for this purpose and tested after the parse.

- *Negative existential constraints* are treated by assigning the value 'nil' to the attribute. The value 'nil' is interpreted as non-existence of the feature and must not appear in the grammar itself.

- *Constraining equations* can be handled as follows*:
We introduce a special term 'C'(Value,Mark), where Value is the value demanded by the constraining equation and Mark is uninstantiated. The definition of the unification is changed, such that a simple value X is treated as a short form of the term 'C'(X,t). Therefore, any unsatisfied constraining equation results in an uninstantiated mark in the f-structure and can easily be detected after parsing.

*this treatment of constraining equations is due to an idea of Jo Calder (personal communication).
In our current implementation a more general method is used which also allows both sides of the equation to denote substructures, but which is omitted here since these cases never occur in realistic grammars.

- *Completeness* of f-structures is tested by existential constraints on the sub-structures required by the semantic form.
We think that the mere existence of a required sub-structure is not enough. For example, verb entries often introduce a partial f-structure for the subject by specifying its number. This should not lead to the acceptance of a sentence without a subject. For that reason we use existential constraints on the 'pred' of a structure to test if it is there.

- *Coherence* of an f-structure is equivalent to negative existential constraints concerning all governable functions (i.e. functions that can appear in semantic forms) that are not required by its semantic form. Introduction of negative existential constraints for all those attributes, as described above, would be a correct but inefficient solution. Instead we use a special mark 'ngf', which closes an f-structure for governable functions, i.e. the definition of 'merge' is extended by an additional test that prohibits the insertion of a governable function after the 'ngf'-mark.

Example: the lexical entry

promised: V, (↑ TENSE) = PAST
(↑ PRED) = 'PROMISE⟨(↑ SUBJ) (↑ OBJ) (↑ VCOMP)⟩'
(↑ VCOMP TO) $=_c$ +
(↑ VCOMP SUBJ) = (↑ SUBJ)

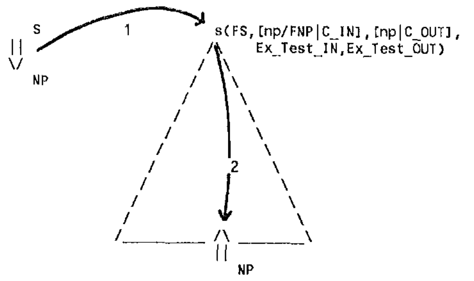
is transformed to

```
v(V, Ex_Tests, [PSUBJ, POBJ, PVCOMP|Ex_Tests]) -->
        [promised],
        {merge(V,[tense = past,
                  pred  = promise(subj,obj,vcomp),
                  subj  = [pred = PSUBJ | RSUBJ],
                  obj   = [pred = POBJ | _],
                  vcomp = [to   = 'C'(+,_),
                           pred = PVCOMP,
                           subj = [pred = PSUBJ | RSUBJ]
                            | _ ],
                  ngf
                   | _ ])}.
```

Treatment of Long-Distance Dependencies

In order to handle long-distance dependencies correctly, LFG provides bounded domination metavariables. The conditions for proper instantiation given by [Kaplan/Bresnan 82] have to be satisfied. They concern:
- The relation between domain roots and controllers
- The one-to-one assignment between domain root and controllee
- The observance of the crossing limit.

Also, bounding nodes, i.e. nodes that are excluded from the control domains of higher nodes, have to be handled correctly.



Treatment of bounded domination metavariables in two steps

The binding of the bounded domination metavariables consists of two steps. The first step, the identification of the domain roots, only depends on the grammar and can be done during the transformation of the grammar rules into PROLOG clauses.
The main job, the assignment between domain root and controllee, is performed as follows:

Each goal has two extra parameters for input and output of a controller list. These lists, which are threaded through all nodes, except the bounding nodes, act as a global stack on which the controllers are pushed. Each element of the stack refers to a node which dominates the current goal, and which is a domain root.
A domain root adds an element to the stack before the parser enters its control domain and removes a receipt after the domain is left. The element that is pushed consists of the class name (eg. [+wh]) of the controller and its actual variable.
If a controllee appears, the stack is searched for the first element with the same class name (for crossing limit n the first n+1 matching elements can be chosen) and replaces it by a receipt. Now the controllee can use the actual variable of the controller.
This treatment resembles the hold list device in the ATN formalism [Winograd 83] a lot, but differs in two important aspects.
- By using unification to establish the correspondence between controller and controllee, information may flow in both directions.
- A controllee does not cause a pop-operation on the stack, but a substitution of an element by a receipt. The checking of the receipt by the domain root ensures that a controllee can only occur within the domain of its domain root.

As an example, the transformations of LFG rules with controller and controllee are given:

$$\begin{array}{ll} NP \rightarrow & e \\ & \uparrow = \Uparrow_{NP} \end{array}$$

```
np(Fnp, CL0, CL1, Ex_Tests, Ex_tests) -->
            [],
            {subst(np/Fnp, np, CL0, CL1)}.
```

$$\begin{array}{lll} S' \rightarrow & NP & \boxed{S} \\ & (\uparrow Q) = \Downarrow^{NP}_{[+wh]} & \uparrow = \downarrow \\ & (\uparrow FOCUS) = \downarrow & \\ & \downarrow = \Downarrow^{S}_{NP} & \end{array}$$

```
s_bar(Fs_bar, CL0, CL1, Ex_Tests0, Ex_Tests2) -->
            np(Fnp, [wh/Q|CL0], [wh|CL1], Ex_Tests0, Ex_Tests1),
            {merge([q=Q, focus=Fnp |_], Fs_bar)},
            s(Fs_bar, [np/Fnp], [np], Ex_Tests1, Ex_Tests2).
```

Treatment of Left Recursion

Definite Clause Grammars do not allow left-recursive grammar rules when interpreted by a top-down parser. This is a serious shortcoming for a natural language system, since many linguistic phenomena can be most naturally described with left-recursive rules (coordination, possessive NPs etc.).
In the theory of formal languages, there exist several algorithms to convert a grammar containing left recursion into a weakly equivalent grammar that does not [Aho/Ullman 77]. But in LFG, the c-structures are essential for the correct evaluation of f-structures, so a transformation must provide a way to get the right interpretation of the functional description.
For a detailed discussion of how this can be achieved for locally left-recursive rules, please refer to [Eisele 85].

Experience with the System

We have implemented two versions of the LFG system, both running on a VAX 11/780. The first version was written by the authors in PROLOG II, using ideas of W.Frey and U.Reyle. It made use of the built-in predicates 'freeze' and 'dif', which give the possibility of delaying subgoals to optimize the evaluation of constraints [Eisele 85].
To improve the flexibility of the user interface, the system was reimplemented in C-PROLOG by Stefan Schimpf and Andreas Eisele. It has been used for the development and testing of different grammars for fragments of English, German [Netter 86] and French, the latter consisting of about 50 grammar rules and more than 200 lexical entries, and turned out to be a useful tool for this purpose.
The performance of the system is quite good for simple grammars with a small amount of nondeterminism. Using the grammar given in [Kaplan/Bresnan 82], parsing the sentence

"I wondered which violin the sonata is tough for her to play on"

needs about 2.3 seconds cpu time (C-PROLOG interpreter).
Yet, we don't expect that our system constructs an efficient parser from an arbitrary grammar mainly for two reasons:
- The complexity of the LFG recognition problem is known to be NP-complete [Berwick 82].
- Our approach to handle nondeterminism by mere backtracking leads to unneccessary duplications of parsing actions.

Whereas the first point is highly questionable as to whether it concerns practical grammars, there are several possibilities to improve the behaviour of the parser.
- Storing intermediate results in a chart could help to avoid multiple parsing of the same constituents and would facilitate error analysis.
- Explicit representation of ambiguities in f-structures (instead of a chronological enumeration) would be a step towards a packaging of local ambiguity.

But in either case, the built-in structure-sharing mechanism of PROLOG could not be used as straightforward a way as in our current system and the definition of unification would have to be considerably more complex.

References

Aho, A.V. and Ullman, J.D., "Principles of Compiler Design", Addison-Wesley, Reading, Mass., 1977.

Berwick, R.C., "Computational Complexity and Lexical-Functional Grammar", ACL Journal, Vol.8, 1982.

Eisele, A., "A Lexical Functional Grammar System in Prolog", LDV-Forum 2/85.

Kaplan, R.M. and Bresnan, J., "Lexical-Functional Grammar: A Formal System for Grammatical Representation" in "Mental Representation of Grammatical Relations", Bresnan eds., MIT Press, 1982

Netter, K., "Getting Things Out Of Order", this volume.

Shieber, S.M., "An Introduction to Unification-Based Approaches to Grammar", Tutorial Session at the 23rd Annual Meeting of the ACL, Chicago, 1985.

Winograd, T., "Language as a Cognitive Process", Vol.1 Syntax, Addison-Wesley, Reading, Mass., 1983.