A LEXICAL FUNCTIONAL GRAMMAR SYSTEM IN PROLOG

A. Elsele
Department of Linguistics
University of Stuttgart
D-7000 Stuttgart

Abstract

This paper describes a system in PROLOG for the automatic transformation of a grammar, written in LFG formalism, into a DCG-based parser of the language. 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 weak points 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, 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 [Bresnan, Kaplan 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 by Bresnan/Kaplan. The input is translated into PROLOG rules, which form a top down parser in definite clause grammar style.

The whole system is written in PROLOG II [Caneghem 82], which allows for delaying subgoals by using the predicates FREEZE and DIF. These features are used to optimise the evaluation of the l-
structures. Equations and constraints associated to a grammar rule are evaluated as soon as the rule is used; and thus wrong parses can be rejected 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 automatically converting these rules to right-recursive ones in such a way that the f-structures remain the same.

Principles of the Parser

Bresnan gives an algorithm in three steps to get the f-structure associated to a sentence. Firstly the constituent structure of the sentence is discovered, using only the context-free parts of the grammar. In a second step, the equations and constraints associated to the grammar rules used are collected and the meta-variables are replaced by real variables. In a third step, the system of equations is solved and the f-structure of the input sentence is found.

Considering that the functional description can be used to reduce the syntactical ambiguities of the input sentence and that the process of solving the equations is independent of their order, parsing the sentence and solving the equations can be done in parallel.

In doing so, there must be the possibility to denote partial f-structures, i.e. f-structures which result from solving some of the equations and which are still to be extended when considering further information.

We represent partial f-structures as an 'open ended' list of pairs:

\[
\{ A_1 = V_1 , A_2 = V_2 , \ldots , A_n = V_n \mid \_ \} 
\]

where the A_i stand for (atomic) attributes and the V_i for the values associated to these attributes. These values are either atomic, terms denoting semantic forms, themselves f-structures or the term 'set(S)' where S stands for an (open ended) list of f-structures denoting a set.

* Although PROLOG II differs from the standard PROLOG syntax and does not allow for writing grammar rules in DCG formalism, I use the syntactic features of CPROLOG for better readability.
An equation that expresses the identity of two f-structures can thus be evaluated by inserting into both structures the features missing with respect to the other, and then unifying the variables that stand for the rest of the lists.

A procedure which performs this action can easily be written in PROLOG:

\[
\begin{align*}
\text{del}(F,[F|X],X) & :- !. \\
\text{del}(F,[E|X],[E|Y]) & :- \text{del}(F,X,Y).
\end{align*}
\]

\[
\begin{align*}
\text{merge}(X,X) & :- !. \\
\text{merge}([A=V1|R1],F2) & :- \text{del}(A=V2,F2,R2), \\
& \quad \text{merge}(V1,V2), \\
& \quad \text{merge}(R1,R2).
\end{align*}
\]

Despite its name, the predicate \text{del} can be used to insert an element at the end of an open ended list if it is not already a member of this list. In both cases, the third argument is instantiated to a list which is identical to the second argument minus the first. The predicate \text{merge} takes two f-structures as its arguments and expands them such that they become identical if they do not contain contradicting values for any attribute at any level: if they do so, the predicate fails and nothing is changed. (The treatment of sets is omitted here for the sake of simplicity.)

Example: The goal \text{merge}([\text{subj}=[\text{spec}=\text{def}, \\
\text{num}=\text{sg}, \\
\text{pred}=\text{girl} | \text{Rsubj1}] | \text{RS1}], \\
[\text{pred}=\text{hand}(\text{subj},\text{obj2},\text{obj}), \\
\text{tense}=\text{present}, \\
\text{subj}=[\text{num}=\text{sg} | \text{Rsubj2}] | \text{RS2}]).

yields the assignments

\[
\begin{align*}
\text{RS1} & = [\text{pred}=\text{hand}(\text{subj},\text{obj2},\text{obj}), \\
& \quad \text{tense}=\text{present}|\text{RS2}] \\
\text{RSubj2} & = [\text{spec}=\text{def},\text{pred}=\text{girl}|\text{RSubj1}]
\end{align*}
\]

Using only the predicate \text{merge} for expressing equality, all LFG rules with equations can be transformed into DCG clauses.
For example

\[ S \rightarrow NP \text{ ('SUBJ)=v} \quad \text{VP ')=v} \]

is translated into the DCG rule

\[ s(S) \rightarrow np(NP), \{ \text{merge([subj=NP],\_),S} \}, \text{vp(S).} \]

**Treatment of Constraining Equations**

After merging two f-structures, they remain identical during the whole further computation, i.e. each extension to one of them also affects the other. This feature can be used, in connection with the coroutines of PROLOG II, to treat constraining equations in a simple and efficient way: goals which test constraints can be 'frozen' to variables that are part of an f-structure. If such a variable is instantiated later (maybe while parsing a different constituent of the input sentence) the goal is 'woken up' and if the test now fails, backtracking is invoked.

In this way all kinds of constraining conditions in LFG formalism and even the tests for completeness and coherence of the f-structures can be evaluated locally - there is no need to 'gather' them in extra variables and test them afterwards.

Specifically, we proceed as follows:

- Existential constraints are treated by inserting the attribute into the f-structure, but the associated value remains variable (if it isn't already known). To the PROLOG variable that represents this value the condition \# nil is bound using 'freeze'. After parsing, all the remaining variables in the structures are set to the value 'nil', so a missing value can be detected. (In general, the violation of an existential constraint cannot be detected sooner, because the parser cannot know if it will still get a value for the attribute).

- Negative existential constraints are treated by assigning the value 'nil' to the attribute. The value 'nil' is interpreted as non-existence of the feature, so the change of the f-structure is
not an essential one. The contradicting case in which both a positive and a negative existential constraint are required, is detected immediately.

- Constraining equations are evaluated using the predicate merge_c, which compares the parts of its arguments that are already known and freezes the further tests to the remaining variables. As soon as a contradiction appears, backtracking is invoked.

- Negative constraining equations are handled by the predicate 'different' which uses the PROLOG II primitive 'dif'. Dif fails as soon as its arguments are unified, but waits if they still contain parts represented by different variables.

- 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 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 means a negative existential constraint concerning all governable functions that are not required by its semantic form. We use the predicate 'ngf', which checks all attributes of the remaining structure as soon as they are built in and fails if they are governable functions.

Since the invocation of 'frozen' tests can interact with the procedure that merges f-structures, the real definition of 'merge' had to be slightly altered.
Definitions of some of the predicates used are given in the appendix.
Example: the lexical entry promised:

\[ V, (** TENSE) = PAST (** PRED) = 'PROMISE (** SUBJ)(** OBJ)(** VCOMP)' (** VCOMP TO) = V (** VCOMP SUBJ) = (** SUBJ) \]
becomes

\[ v(V) \rightarrow [\text{promised}], \]
\[
\text{merge}(V, [\text{tense}=\text{past}, \]
\quad \text{pred}=\text{promise}([\text{subj}, \text{obj}, \text{vcomp}], \]
\quad \text{subj}=\{\text{pred}=\text{PSUBJ}|\text{RSUBJ}, \]
\quad \text{obj}=\{\text{pred}=\text{POBJ}|\_], \]
\quad \text{vcomp}=\{\text{to}=\text{PLUS}, \]
\quad \text{pred}=\text{PVCOMP}, \]
\quad \text{subj}=\{\text{pred}=\text{PSUBJ}|\text{RSUBJ}|\_}\}
\]
\]
\[
\text{freeze}(\text{PLUS}, \text{PLUS++}), \]
\[
\text{freeze}(\text{PSUBJ}, \text{PSUBJ}=\text{nil}), \]
\[
\text{freeze}(\text{POBJ}, \text{POBJ}=\text{nil}), \]
\[
\text{freeze}(\text{PVCOMP}, \text{PVCOMP}=\text{nil}), \]
\[
\text{ngf}(R2) \}.
\]

where the first freeze subgoal ensures that, whenever PLUS gets instantiated, it must be set to +, the other freeze subgoals ensure that the preds of the structures demanded by the semantic form must not be set to nil (completeness) and the ngf constrains the further expansions of the f-structure to attributes that are not governable functions (coherence).

Treatment of Long-Distance Dependencies

To handle long-distance dependencies correctly, the conditions for proper instantiation given by Bresnan/Kaplan have to be satisfied. They concern:

- The relation between the domain roots and the long distance controllers.

- The one-to-one assignment between domain root and controller

- 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.
One of the tasks, the identification of the domain roots, only depends on the grammar and is performed during the transformation of the grammar rules into PROLOG clauses.

The main job, the assignment between domain root and controller is performed as follows:

Each goal has two extra parameters for input and output of a controller list. These lists 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 one 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 controller 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). Now the controller can use the actual variable of the controller.

To check the correct assignment between domain root and controller the stack element is replaced by a 'receipt'. The domain root tests this when it removes the stack element.

As an example, the transformation of an LFG rule with controller and of an entry for a controller are given:

\[
NP \rightarrow \text{eps} \quad ^* = (<= \text{np}).
\]

\[
\text{np(CLO,CL1,Fnp)} \rightarrow \{\},
\]

\[
\text{s_bar(CLO,CL1,Fs_bar)} \rightarrow \text{np([wh/Q|CL0],[wh|CL1],Fnp)},
\]

\[
\text{bounding S} \quad ^* = v.
\]
Treatment of Left Recursion

Definite Clause Grammars do not allow left-recursive grammar rules to be used because they lead to loops in the 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 an equivalent grammar that does not.

But equivalency means here that the same language is generated with different assignment of c-structures to the input sentences.

In LF 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.

At the moment, our system performs an automatic conversion of locally left-recursive rules into rules which are right-recursive and which produce the same f-structures.

The method is best shown with an example:

The rules

\[
A \rightarrow A \ B, \quad A \rightarrow C.
\]

can be transformed to the equivalent grammar

\[
A \rightarrow A_1 \ A_2, \quad A_2 \rightarrow B \ A_2, \quad A_1 \rightarrow C, \quad A_2 \rightarrow [\ ].
\]

This transformation can be performed for every local left recursive symbol in a one-to-one manner, where only the two extra rules on the left are needed. But now the c-structure of an input string \(C \ B \ B \ B\) is completely different:

\[
\begin{align*}
\text{A} & \quad \text{A} \\
\text{A} & \quad \text{A} \\
\text{A} & \quad \text{A} \\
\text{A} & \quad \text{A} \\
\text{A} & \quad \text{A} \\
\text{A} & \quad \text{A} \\
\end{align*}
\]
In the old grammar one could say that each A node takes the f-structure(s) of its subordinate A and B or C node(s) and uses them to build its own f-structure.

In the new grammar, the A1 node behaves just as the A node dominating the C node in the old grammar, but the A2 nodes can be considered as taking the f-structure of the left-sister A1 node or the dominating A2 node and that of the subordinate B node to build its own f-structure.

If the f-structures are evaluated this way and passed downward, then the f-structure belonging to the A node will be assigned to the A2 node which expands to the empty string.

We have to provide an extra parameter for passing the f-structure of A through all the A2 goals.

The following example shows the above grammar augmented with variables for f-structures and with goals which evaluate these f-structures:

\[
\begin{align*}
a(FH) \rightarrow & \ a(FA), \{goal\_a(FH,FA)\}, \\
& b(FB), \{goal\_b(FH,FB)\}. \\

a(FH) \rightarrow & \ c(FC), \{goal\_c(FH,FC)\}.
\end{align*}
\]

This would be converted to the right-recursive DCG program that yields the intended f-structures:

\[
\begin{align*}
a(FH) \rightarrow & \ a1(F1), a2(F1,FH). \\
a2(FA,F) \rightarrow & \ [], \{merge(FA,F)\}. \\
a2(FA,F) \rightarrow & \ \{goal\_a(FH,FA)\}, \\
& b(FB), \{goal\_b(FH,Fb)\}, \\
& a2(FH,F). \\
a1(FH) \rightarrow & \ c(FC), \{goal\_c(FH,FC)\}.
\end{align*}
\]
Experience with the System

The LFG system, which has been written by Jochen Doerre and myself, using ideas from Werner Frey and Uwe Reyle, runs on a VAX 780 and has been tested with different grammars of English, German and French, the latter consisting of about 30 grammar rules and over 200 lexical entries. Our expectations concerning the parse times were not quite fulfilled.

We see mainly three reasons for long parse times:

- The language PROLOG II, which was originally designed to run on an Apple and was afterwards adapted to a VAX, does not seem appropriate for high-efficiency language analysis.

- The grammars contain many optional constituents and constituents that can be repeated arbitrarily often, which are handled automatically by introducing new nonterminal symbols and empty productions for them. This leads to a high amount of backtracking in the top-down parser, which does not regard the input string before it has expanded its expectation down to the lexical level.