SIMPLE PARSER FOR AN HPSG-STYLE GRAMMAR IMPLEMENTED IN PROLOG

Karel Oliva*
Lingustic Modelling Laboratory,
Coordination Centre
for Computer Science and Computer Technology,
Bulgarian Academy of Sciences,
acad. G. Bonchev st. bl. 25A,
BG - 1113 Sofia,
Bulgaria

Abstract:
This paper describes basic ideas of a parser for HPSG style grammars without LP component. The parser works bottom-up using the left corner method and a chart for improving efficiency. Attention is paid to the format of grammar rules as required by the parser, to the possibilities of direct implementation of principles of the grammar as well as to solutions of problems connected with storing partly specified categories in the chart.

1. Representation of Grammar Rules for the Parser

The Head-driven Phrase Structure Grammar (HPSG) blurrs the distinction between rules of the grammar and the structures they generate. Put shortly, the matter is that "structures" and "rules" in HPSG differ solely in the level of abstraction over the linguistic material they describe. A "structure" describes some very concrete piece of this material (e.g., a sentence) and, hence, embodies no abstraction; a "rule", on the other hand, presents by itself a prototype of a set of structures. Since in HPSG categories are understood as bundles of features ("attribute"="value" pairs) , the "structure"/"rule" dichotomy is reflected by the fact that the rules can contain variables as values of attributes of some features while the structures must be always fully specified or that the rules can miss some (otherwise possibly obligatory) features altogether. Constraints restricting or binding together permitted values of the attributes can be associated with the rules. Naturally, different levels of abstraction can be introduced among the rules as well, which allows for capturing different levels of generalization over the linguistic data described.

On the highest level of abstraction, the parser can deal with two types of rules: in the first type, the values of variables occurring in the rules are bound by constraints, in the second type no constraints occur. In order to support simultaneously an easily legible notation and a reasonable computer implementation of these two types of rules, two Prolog operators are defined, each describing one rule type.

```
:- op(1200,xfx,is_a_rule_if) .
:- op(1200,xf,is_a_rule) .
```

The first of the two is an infix operator describing the rules containing additional constraints; the rule itself should stand in front of the operator, the constraints should follow it, separated from each other by commas ",". The second one is a postfix operator describing the rules without any constraints.

The inventory of types of rules may be arbitrarily broadened. All that is necessary for this purpose is just adding operator declarations and, possibly, also implementing feature inheritance principles corresponding to the newly introduced rule type(s). This is important because it provides for bounding the application of the principles to the whole rule types and makes thus obsolete the explicit stipulation of feature sharing among respective categories in each rule, which is still the case in many current parsers.

Two examples of the rule format for the parser are shown in the following: it is to be remembered that in HPSG, as well as in all other theories accepting the X-bar convention, a central role among the daughters in a rule is played by the head-daughter - because of this, the head-daughter is specially marked, which provides, e.g., for application of the Head Feature Principle.

Ex.1: - the standard "S ---> NP VP" rule can appear in the following form (with obvious meanings of the predicates "concatenation" and "agreement"):

```
    [phonology=S_Phonology,
     d_trs=[d_tr=[cat=n,
                  bar=two,
                  phonology=NP_Phonology,
                  morphology=NP_Morphology ],
            head_d_tr=[cat=v,
                       bar=two,
                       phonology=VP_Phonology,
                       morphology=VP_Morphology]]]
is_a_rule_if
 concatenation(NP_Phonology,VP_Phonology,S_Phonology),
   agreement(NP_Morphology,VP_Morphology) .
```

Ex.2: - the rule "NP ---> Det NP": note the fact that the phonology of the mother can be expressed without invocation of the "concatenation" predicate (since determiners consist of one word only) and the agreement is expressed directly in the rule by coindexing the features "number" in both daughters

```
    [bar=two,
     phonology=[Det_Phonology|NP_Phonology],
     d_trs=[d_tr=[cat=det,
                  bar=zero,
                  phonology=[Det_Phonology],
                  morphology=[number=Number]],
            head_d_tr=[cat=n,
                       bar=one,
                       phonology=NP_Phonology,
                       morphology=[number=Number]]]]
  is_a_rule .
```

2. Representation of Categories in the Parser

As follows from the examples, the notation adopted for categories in rules is the one describing them as (Prolog) lists of features. Keeping such kind of representation also in the underlying mechanism of the parser would be, however, quite unfelicitous a decision. The main problem consists in the fact that the parser working bottom-up may discover certain features of already parsed (sub)structures only later in the parsing process (so to say, only when it gets "higher in the tree", with regard to the way the parsing proceeds). These features are to be, then, incorporated into the already parsed structures. An elegant solution of this problem was proposed in (Eisele and Dörre,1986) and adopted in the parser described. Syntactic categories are represented in the parser internally in a way slightly different from their representation in the grammar: all categories (including those used as values of features of other categories) are represented as "open-ended lists": each internal representation of a category is a list having a certain number of instantiated elements at its beginning and an uninstantiated "tail". The main idea standing behind this kind of representation is that any feature to be discovered (and added to the category) only later in the parsing process can be now added as the "first member" of the uninstantiated "tail", which task is easy to perform provided that the "tail" is still accessible (e.g., if the free "tails" of categories subject to feature inheritance principles are shared logical variables). Converting categories from one kind of representation to the other one is performed by a two-argument predicate "perestroika" (used below).

The representation described also supports a simple implementation of unification of categories (see Eisele and Dörre, op. cit. for more detail); in the folowing, unification of two categories is presupposed to be performed by a two-argument predicate "unify".

3. The Parser

The main idea of the parsing method used in the BUP parser (Matsumoto et al,1983) being the starting point of the system described is that a rule is to be triggered only after its left corner has been found (i.e. it has been supplied by lexical scan, in the case of lexical categories, or it has been properly parsed). The left corner of a rewriting rule is the leftmost symbol on its right-hand side - the name stems from depicting the rule as a local tree it generates. After a category is parsed or supplied by lexical scan, one of the grammar rules having this category as its left-corner is selected, the sisters of the left corner in this rule are tried, and if all of them are succesfuly parsed, the mother category of the rule is declared to be parsed and the whole process, using the mother as a left corner, is repeated "on a higher level". If any failure occurs, backtracking is invoked. Thus, the parsing process is data-driven - the rules of the grammar are selected in accordance with the symbols scanned in the input. From the viewpoint of efficiency, this is important mainly for the so-called "free-word-order" languages. Mentioning this, it should be further recalled that the performance of BUP is further improved by storing all the information about all subtasks that have been already tried (successfully or unsuccessfully), which avoids repetitive computations of the parses that have been performed or that have been proved impossible to perform in the preceding steps of the analysis.

For the purposes of the implementation of the parsing process, it is necessary to extend the notion of the "left corner" to its reflexive and transitive closure. The transitive closure inductively states that for all triples of categories X,Y,Z such that X is a left corner of Y and Y is a left corner of Z , X is also a left corner of Z. The reflexive closure finishes the picture by saying that any category is a left corner of itself.

Given the previously described basic philosophy of parsing, the process can be implemented in Prolog by means of two predicates performing the two tasks informally mentioned in the preceding paragraphs:

- the predicate "parse", parsing a given (expected) category from (a prefix of) the input string
- the predicate "is_a_left_corner", linking the left corner category with the goal (expected) category in the parsing process.

However, before these predicates can be explained in more detail, it is necessary to make several remarks explaining the way the processing of complex categories has been built into the system.

First, the usual equality ("=") of two categories was replaced by their unification, i.e. on all spots where equality of two categories - expressed either directly, in the form of an equation, or indirectly, by variable sharing or otherwise - occured in the original BUP, it had to be replaced by a call of the predicate "unify".

Second, in the predicates storing or retrieving the information about the (un)successfully performed parsing subtasks, the categories must be "frozen" exactly in the state when this subtask was started: problems would occur if the "stored" categories include free variables ("information holes") as values of some features, which variables might be matched by any real values in the moment of search for the information about previously performed parsing tasks - such a matching, however, would be incorrect, since what is required is a real identity of the subtasks. (The same holds also the other way round, i.e. problems of exactly the same nature would occur also if the stored value were instantiated and the current one were a free variable.)

The aforementioned identity of subtasks, however, requires the identity of (some of) the stored categories only, not the identity of the lists representing them, i.e. what really matters is the identity of features, but not of their order. This identity of "frozen" categories (represented as "usual" Prolog lists) is checked by the predicate "identical_categories".

Now at last, the definitions of the predicates "parse" and "is_a_left_corner" can be given; the supporting predicates are either elucidated in the preceding text or are given (hopefully) self-explaining names, which should hold also for the arguments. The difference between the "frozen" categories represented as usual Prolog lists and those represented as "open-ended" lists is reflected in the variable names standing for the respective types: the "open-ended" categories are always marked as "Real" categories, the other ones never bear such marking.

```
%   PARSE(
%      "Frozen"_Goal_Cat,
%      [Real_Goal_Cat,Structure],
%      Input_String,Rest_String   )

/* Checking whether parsing the current Real Goal Ca-
tegory from some prefix of the Input String has been
tried (either successfully or not) in the preceding
steps of the parsing process. */
parse(Goal_Category,[Real_Goal_Category,Structure],
      Input_String,Rest_String                       )
 :-
    ( already_parsed(Asserted_Goal_Category,_,
                     Input_String,_             ),

      identical_categories(Goal_Category,
                           Asserted_Goal_Category) ;

      cannot_be_parsed(Asserted_Goal_Category,
                       Input_String           ),

      identical_categories(Goal_Category,
                           Asserted_Goal_Category),

      !, fail                                        ),

    !,

    already_parsed(Asserted_Goal_Category,
                   [Real_Goal_Category,Structure],
                   Input_String,Rest_String       ),

    identical_categories(Goal_Category,
                         Asserted_Goal_Category) .
```

```
/* The following clause describes parsing of a cate-
gory with no daughters (category immediately dominat-
ing an empty string) */
parse(Goal_Category,[Real_Goal_Category,d_trs=[]],
      String,String                               )
  :-
    /* rule having no daughters is to be found in the
grammar */
    find_rule_to_be_used(Real_Goal_Category,
                         Constraints_Of_Rule),

    call(Constraints_Of_Rule),

    assertz(already_parsed(
              Goal_Category,
              [Real_Goal_Category,d_trs=[]],
              String,String              )) .

/* The following clause describes parsing of a cate-
gory dominating a non-empty terminal string */
parse(Goal_Category,[Real_Goal_Category,Structure],
      [Word_Form|Rest_Input_String],Rest_String    )
  :-
    lexicon(Word_Form,Word_Form_Category),

    perestroika(Word_Form_Category,
                Real_Word_Form_Category),

    is_a_left_corner(
      [Real_Word_Form_Category,d_trs=[]],
      [Real_Goal_Category,Structure],
      Rest_Input_String,Rest_String       ),

    assertz(already_parsed(
              Goal_Category,
              [Real_Goal_Category,Structure],
              [Word_Form|Rest_Input_String],
              Rest_String                  )) .

/* Asserting information about the impossibility of
parsing certain categories from certain strings */
parse(Goal_Category,_,Input_String,_)
  :-
    ( already_parsed(Goal_Category,_,Input_String,_) ;

      assertz(cannot_be_parsed(
                Goal_Category,Input_String)) ),

    !, fail .


%        IS_A_LEFT_CORNER(
%           [Real_Left_Corner_Cat,Structure],
%           [Real_Goal_Cat,Structure],
%           Input_String,Rest_String          )

/* reflexive closure of the relation "being a left
corner"*/
is_a_left_corner(
  [Real_Left_Corner_Category,
   Real_Left_Corner_Category_Structure],
  [Real_Goal_Category,Real_Goal_Category_Structure],
   String,String                                  )
  :-
    unify(Real_Left_Corner_Category,
          Real_Goal_Category        ) .

/* transitive closure of the relation "being a left
corner" */
is_a_left_corner(
  [Real_Left_Corner_Category,
   Real_Left_Corner_Category_Structure],
  [Real_Goal_Category,Real_Goal_Category_Structure],
  Input_String,Rest_String                        )
  :-
    /* a rule having the current left corner category
as its left corner is to be found */
    find_rule_to_be_used(
      Real_Left_Corner_Category],
      Left_Daughter_Marking,
      Right_Sisters_List_From_Current_Rule,
      Mother_Category_From_Current_Rule,
      Constraints_Of_Rule,
      Type_Of_Rule                        ),

   /* all the right sisters have to be parsed */
    parse_right_sisters(
      Right_Sisters_List_From_Current_Rule,
      Real_Right_Sisters_List,
      Input_String,Intermediary_String,
      Type_Of_Rule                        ),

     call(Constraints_Of_Rule),

    /* the mother category from the rule must comply
with the feature inheritance principles relevant for
the Type_Of_Rule */
     feature_inheritance_principles_concerning_mother(
       [Real_Left_Corner_Category
       |Real_Right_Sisters_List],
       Mother_Category_From_Current_Rule,
       Real_Mother_Category,
       Type_Of_Rule                                 ),

   /* the mother itself (?herself?) is used as a left
corner, which repeats the process on a higher level */
  is_a_left_corner(
    [Real_Mother_Category,
     d_trs=[Left_Daughter_Marking=
              [Real_Left_Corner_Category,
               Real_Left_Corner_Category_Structure],
            |Real_Right_Sisters_List              ]],
    [Real_Goal_Category,Real_Goal_Category_Structure],
    Intermediary_String,Rest_String                 ).
```

The whole parsing process is started by asking the conjunction of goals

```
   ?- parse(TOPMOST_CATEGORY,Intermediary_Result,
            INPUT,[]                           ),
      perestroika(RESULT,Intermediary_Result) .
```

where the "RESULT" is the only output argument, namely, the resulting structure of the parse, the TOPMOST_CATEGORY is a skeleton category (represented as a "usual" Prolog list) of the expected result (most often, something like "[cat=sentence]" or "[cat=v,bar=two]" etc.), i.e. a category which is expected to unify with any result of the parse, the INPUT_STRING is the input string represented as a Prolog list of wordforms and the empty string "[]" is the expected rest of the input string after the parsing process finished.

Bibliography:
Eisele A. and J. Dörre: A Lexical Functional Grammar System in Prolog, in: Proceedings of Coling '86, Bonn
Matsumoto Y. et al.: BUP - A Bottom-Up Parser Embedded in Prolog, in: New Generation Computing vol.1, 1983
Pollard C. and I.Sag: Information Based Syntax and Semantics. vol.1: Fundamentals, CSLI Lecture Notes No. 13, CSLI, Stanford, California 1987

*since 1st April 1990:

Lehrstuhl für Computerlinguistik
Universität des Saarlandes
Im Stadtwald
D-6600 Saarbrücken
(West) Germany