Lekta: A tool for the development of efficient LFG–based Machine Translation systems

J. Gabriel Amores
Departamento de Lengua Inglesa
Universidad de Sevilla
e-mail: gaby@fing.us.es

Jose F. Quesada
Centro Informatico Cientifico de Andalucia (CICA)
e-mail: josefran@cica.es

1 Introduction

In this paper we describe Lekta, a tool for the development of efficient LFG–based Machine Translation systems. The paper is organized as follows:

Part I describes the overall architecture of the tool. Three layers are distinguished in order to preserve expressive power without losing efficiency. The lower level –the "translation kernel"– contains the following submodules: lexical database manager, parser, unifier, transfer and generator. The intermediate level ("control and configuration") is concerned with the compilation of specification languages, trace, output, statistics and setup. Finally, the top level ("specification languages") defines a set of specification languages for the lexicon and grammar, transfer and generation rules. The tool is written in C and assumes a classical transfer–based approach to MT. From a functional point of view, the tool allows the simultaneous manipulation of several languages. Configuration commands establish which languages are used as source and target.

This section also describes some of the computational strategies used in the translation kernel. Special attention is devoted to the parser and the unifier. The parser is based on a bidirectional, bottom-up and event-driven strategy. We use a constructive approach to unification. Efficiency is achieved by a special memory organization model which takes into account the characteristics of the data structures involved in the translation process. We offer statistics with artificial grammars and lexicons to show how the tool would behave in a real application context.

Part II describes the specification languages in detail. For each natural language involved, we can define an analysis grammar and lexicon, a set of transfer modules (from source language to a number of target languages) and a generation grammar and lexicon. The specification of the lexicon allows the use of macros, lists, disjunction, negation, etc. Grammar rules follow the classical LFG notation. The functional equations associated with each rule are augmented with a control

language which allows IF-THEN-ELSE constructions, logical, relational and mathematical operators, string and lists functions such as MEMBER and CONCAT, and specific functions for the control of f–structure wellformedness (COMPLETENESS and COHERENCE). The transfer module (from source language f–structure to target language f–structure) allows a uniform definition of structural and lexical transfer rules. Each rule may be associated with conditions and actions. Conditions are triggered according to the input f–structure and actions involve the manipulation of the resulting f–structure by means of specific functions such as NOTRANSFER, TRANSFERAS and overwrite. Generation rules (from target f–structure to target c–structure) assign a c–structure to the input f–structure depending on the attributes present.

2 Overall Architecture

This section outlines the overall architecture of the system.

2.1 Overall Architecture of the System

Three layers are distinguished in order to preserve expressive power without losing efficiency (Fig. 1). The lower level –the "translation kernel"– contains the following submodules: lexical database manager, parser, unifier, transfer and generator. The intermediate level ("control and configuration") is concerned with the compilation of specification languages, trace, output, statistics and setup. Finally, the top level ("specification languages") defines a set of specification languages for the lexicon and grammar, transfer and generation rules. The tool is written in C and assumes a classical transfer–based approach to MT. From a functional point of view, the tool allows the simultaneous manipulation of several languages. Configuration commands establish which languages are used as source and target.

2.2 Lexical Analysis

Feature structures may be considered one of the standards of current NLP.

More recently, a number of formalisms have incorporated typed or sorted f–structures [3]. Among them are TDL [9], ALE [4], CUF [5] and TFS [6], [7].

Formally speaking, sorted feature terms have an expressive power equivalent to first–order terms in Prolog [11]. However, the flexibility and representational adequacy of feature structures, either typed (HPSG, GPSG) or untyped (DCG, PATR–II, LFG), make them more suitable for the description of linguistic knowledge.

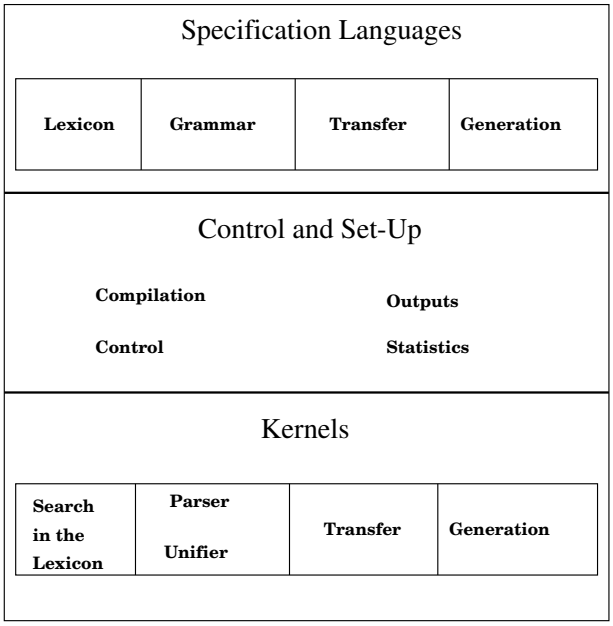

Fig.1. Overall Architecture of the System

This has motivated the implementation of tools that translate from feature structures into Prolog terms[14]. Examples of such systems are CLE [1], ALEP [2] and ProFIT [8].

Nevertheless, most of the systems and tools mentioned above (whether designed for the manipulation of Prolog terms, f–structures or typed f–structures) present some problems of computational efficiency. They have focussed on aspects related to the syntax and semantics of the models, paying less attention to the computational model itself, especially with regard to storage and retrieval. Usually, the computational model is of little importance if we are developing prototypes with a reduced number of lexical entries. The scenario changes dramatically when the goal is the implementation of a real system [16], where the size of the lexicon may condition the feasibility of the overall system.

In this paper we present a computational model aimed at the efficient manipulation of large knowledge bases which use f–structures as their theoretical backbone [12].

2.3 LektaKb: A Novel Approach for the Efficient Storage of Very Large Knowledge Bases

Within the field of knowledge engineering, the problems that arise in NLP as a result of the growth of the lexical database may be understood as a bottleneck in the communication between the inference engine (IE) and the knowledge base (KB).

As a starting point, it is usually required that the KB specification module satisfies a series of properties such as representational adequacy, modularity, high level of abstraction and expressive power.

These requirements are necessary in order to satisfy criteria of adequacy in the interface between the system and the expert knowledge engineer. Unfortunately, it is also the case that those requirements are usually in contradiction with other properties which would allow an efficient interface between the IE and the KB.

Both sets of properties do not seem to be compatible with efficient storage and use of the KB, where we have to take into account low–level computational issues pertaining to the physical architecture of the computer, its operating system and the characteristics of the compiler.

All these problems arise as a consequence of using a "direct" or "flat" model in the design of the interface between the KB and the IE. Instead, we propose a model based on a layered architecture.

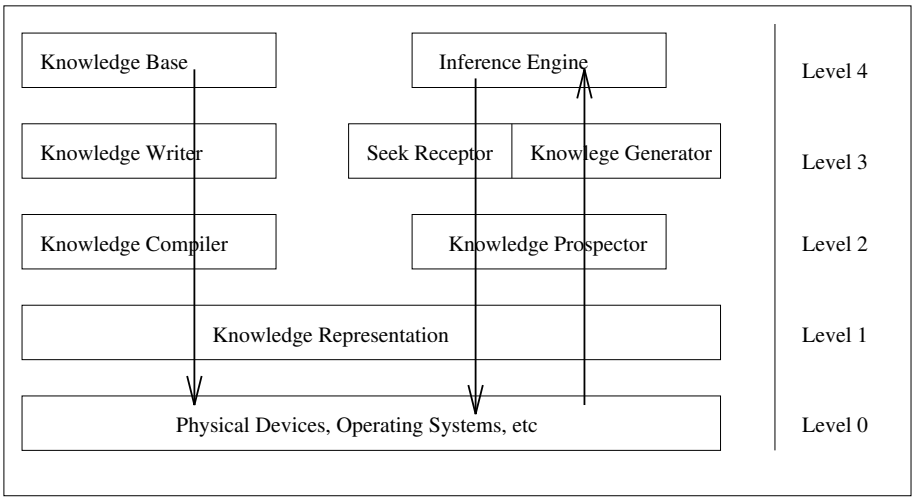

Fig.2. A Layered Architecture between KB and IE

In Fig. 2, the *Knowledge Writer* module defines the KB specification language. Its task is to provide the necessary expressive power. This is achieved through the use of semantically very powerful constructions which enhance the representational adequacy.

The *Seek Receptor* and *Knowledge Generator* modules serve as query languages. Their immediate goal is the achievement of inferential adequacy in the knowledge constructions returned to the IE.

The interface between levels 2 and 3 consists of flat terms. Since their expressive power is equivalent to that of feature structures [11] they are able to convey the same type of information as in level 3. However, limitations relative to representational and inferential adequacy disfavor their use in the interface between levels 3 and 4. As a result, the *Knowledge Writer* and *Knowledge Generator* act as rewrite systems between complex structures and flat terms and viceversa.

The *Knowledge Compiler* module is primarily aimed at securing an optimal logical model of representation. From this model, the *Knowledge Prospector* will be able to perform an efficient use and retrieval of the KB. In sum, level 2, as a whole, will be in charge of use and storage efficiency.

Finally, level 1 performs the actual storage and retrieval of the data. The interface between levels 1 and 2 makes use of packets with a very simple format, whose basic constituents are strings of characters. Level 1 aims at obtaining the best constituents adequacy possible between the logical model in level 2 and the physical properties of the devices (in level 0) involved in storage and retrieval.

The *Knowledge Compiler*, *Knowledge Representation* and *Knowledge Pros-pector* modules use two types of techniques in order to achieve usage and storage efficiency. The first consists of a tetra–dimensional logical organization of the data which adapts the representational model to access probabilities. The second technique incorporates a special type of binary tree (improved binary trees with vertical cut).

2.4 Lexical Analysis Performance

In order to test the efficiency of the lexical component of Lekta (called Vtree [13]) we generated a series of artificial lexicons automatically.

The first column in the table below indicates the size of the lexicon; the second, times taken in loading the lexicon (`VtUse()`); and the remaining columns show information about the search operation (`VtSeek()`) for a total of 10,000 searches.

Records	Vtree: Knowledge Prospector			
	VtUse	VtSeek: T [1]	VtSeek: T/R [2]	VtSeek: R/S [3]
$1(2^0)$	0.0029	1.245	0.00012	8030
$4(2^2)$	0.0029	1.462	0.00014	6840
$16(2^4)$	0.0039	1.801	0.00018	5550
$64(2^6)$	0.0029	1.896	0.00019	5273
$256(2^8)$	0.0039	2.502	0.00025	3996
$1024(2^{10})$	0.0039	4.992	0.00049	2003
$4096(2^{12})$	0.0029	6.679	0.00066	1497
$16384(2^{14})$	0.0029	8.008	0.00080	1249
$65536(2^{16})$	0.0039	10.889	0.00108	918
$262144(2^{18})$	0.0019	12.059	0.00120	829
$1048576(2^{20})$	0.0029	13.166	0.00131	760

Apart from the absolute values of the times obtained in the test, it is interesting to observe that the performance of the `VtUse()` function is independent of the number of records.

A system with a complexity of *O(log(N))* would have multiplied the times of the last database for 20, with respect to the first database. On the contrary, we may see that our results are of the order *O(log(log(N)))*, or more precisely, *2*log(log(N))*.

The same tests have been tried leaving all the management to Prolog and then using Prolog and Vtree concurrently [13].

[1] Total time in seconds.

[2] Time per record, in seconds.

[3] System's performing rate, measured in records compiled per second.

Records	Quintus Prolog			Quintus Prolog with Vtree		
	CT [4]	M [5]	PT [6]	CT (Vtree)	M (Prolog)	PT
$1(2^0)$	0.050	308	0.000	0.050	800	0.007
$4(2^2)$	0.034	436	0.000	0.050	800	0.007
$16(2^4)$	0.133	1,196	0.001	0.080	800	0.010
$64(2^6)$	0.400	4,204	0.002	0.150	800	0.020
$256(2^8)$	1.600	15,620	0.005	0.400	800	0.020
$1024(2^{10})$	6.284	63,076	0.022	1.690	800	0.040
$4096(2^{12})$	25.284	247,436	0.079	8.140	800	0.053
$16384(2^{14})$	102.600	1,145,868	0.300	39.960	800	0.047
$65536(2^{16})$	432.100	4,738,392	1.215	171.760	800	0.063
$262144(2^{18})$	1,764,783	17,783,352	4.709	803.750	800	0.070
$1048576(2^{20})$	***[7]	***	***	4,725,207	800	0.080

2.5 Parsing

The efficiency and robustness of a parser may be said to depend, among others, on three factors: a) the parsing algorithm itself; b) the mathematical model which serves as a base for the representation of grammatical relations, and c) the computational model or data structures which represent the objects to be manipulated by the parser.

In this section we present original ideas with respect to those three criteria. Our basic goal is the design of a parsing kernel for real time applications in natural language processing. Therefore, we propose a complete framework which includes: first, a new parsing technique based on a bottom-up, bidirectional and event-driven parsing strategy; second, a very well found formal definition of a set of predicates and relations that serve as the mathematical background of the parser; and third, a specific computational model that takes into account the memory organization for efficient storage of compiled CFG, as well as the data model used during parsing, that we have named multi virtual-trees. All these components have been defined in an completely unified model [13].

2.5.1 Bottom-Up Parsing with Top-Down Predictions

Bottom-up parsing enriched with top-down predictions is one of the most common strategies in the design of parsers for CFG. Bottom-up models have been defended based on their efficiency. The additional incorporation of top-down predictions improves the overall efficiency of the parser. As a result, we could say that these parsers are controlled simultaneously by the data (the input string) and the goal (the grammar).

[4] Total compilation time in seconds.
[5] Memory needed to store the knowledge base, in bytes.
[6] Average time in analysing a string of five words.
[7] The system used up all memory and could not compile this base.

We present a parser which incorporates both strategies in a novel way. The core of the bottom-up component is a bidirectional generator of analysis events, whose strategy is theoretically founded in a chart model. In our case the events will take on the roles of both active and inactive edges in a chart. Nevertheless, a pure chart generates during analysis an overhead of edges which reduces the overall efficiency of the system. To avoid this, a top-down component will act upon the bottom-up generator imposing strong restrictions. This will result in the elimination of edges (events) without any possibility of success.

2.5.2 Mathematical Foundations

The top-down predictor component must be mathematically sound, so that the elimination of an event must be an unfailing decision. With this goal, we define a set of relations and predicates in the domain of context free grammars which will serve as the basis for the definition of top-down predictions.

In regards to the mathematical model, we use the specification of a CFG to define the following predicates and relations: root symbols, epsilon symbols, left and right partial derivability, left and right primary adjacency, left and right adjacency, left and right partial self-derivability, left-most symbol and right-most symbol.

Associated with this mathematical model we present a computational organization, that we have named Q-memory, which improves the operations involved in the manipulation of symbols and productions. Specially, this model has been designed to avoid the operations of string character comparison.

2.5.3 Bidirectional Parsing and the Plane of Analysis

The choice of the mathematical model above is mutually dependent on the parsing algorithm. Basically, it consists of a bottom-up, bidirectional and event-driven parser. The events behave as analysis objects which are triggered by coverage tables. Moreover, there is also a hierarchy of filters which use the information available from the "plane of bidirectional analysis" and the tables of derivations and adjacencies in order to reject multiple events in the first stages of analysis.

For deterministic analyses, the parser actually manipulates a "surface of analysis" which evolves in time, until it becomes the root node in the grammar. For ambiguous grammars, it is necessary to refer to a "plane of analysis", leading to the problem of how to define structures computationally efficient to store so much information.

2.5.4 Event Driven Parsing and Multi Virtual-Trees

We propose the creation of multi virtual-trees, a novel approach to the notion of analysis forests, which is based on virtual relations among the components of the forest and on the separation of tree skeletons from the contents of each node.

The basic goal of our proposal has been to achieve a complete independence between the nodes which make up a tree or a forest of analyses and the structures and algorithms manipulated by the parser. This independence has been achieved

by the use of MvtNod (node of a multi virtual-tree) and MvtCaD (collection and difussion of events in an multi virtual-tree) structures in such a way that when the analysis finishes we do not obtain a compact tree but a set of MvtNod's, each of which knows one or more possibilities of analysis. This motivates the use of the adjective "virtual" when describing the model. What are actually virtual are the relations between the MvtNod's, and in turn, these relations are transparent for the parsing algorithm, which only takes into account the events which depart from or arrive at each MvtCaD.

The basic idea underlying the parser is that the analysis is an operation internal to an MvtCaD. As a consequence, we will be able to delete, merge, transfer or run an event. Each time that an event has been modified the MvtCaD's affected will be marked using some flags in the MvtCaD structure.

2.5.5 Experimental Results

These ideas have been implemented in C on a medium-size workstation and have yielded results of between 1000 and 5000 words analyzed per second, for the most complex grammars found in the specialized literature.

The experimental results have shown a real complexity of the order O(n) depending on the length of the input string for grammars including recursive constructions and local and non-local dependencies.

Next, we show the predicted model obtained for each type of grammar. The dependent variable T is the time used for the complete analysis (in seconds) and the factor used, W, has been the length of the input string (number of words).

- Recursive Structures
  $T = -81E - 5 + 0.0003 * W$
- Local Dependencies
  $T = -0.005 + 0.0012 * W$
- Non Local Dependencies
  $T = -0.004 + 0.0009 * W$

2.6 Unification

In NLP, unification is usually described as a computational sink. During the last years, research has concentrated in elucidating the reasons of such a problem. Some authors have identified processes such as early copying, over copying or redundant copying. We propose a novel model based on constructive unification, that avoids all copying problems. The general constructive model can be expanded using different techniques like strategic unification, sub-structure sharing or post-copy in order to accomodate to different situations.

The algorithm permits the incorporation of a strategic unification model, using probabilistic information about previous successes and failures.

The constructive model may be viewed as a family of unification algorithms based on sub-structure sharing or post-copy techniques [13].

2.7 Generation

Our basic goal was the design and implementation of a robust and efficient generation model. The generation component takes as input the f–structure resulting from transfer and returns as output the corresponding c–structure and the string of words associated with that tree. Decisions such as target lexical choice and structural organization were taken during the transfer stage.

From a computational point of view, the generation process may be understood as a recursive function (GENERATE) in charge of generating the non–terminal nodes. Terminal nodes are generated by a SYNTHESIS function which is called when no more substructures are necessary. From the linguistic point of view, we have to solve at least two problems: first, that all the information in the input f–structure has been consumed and second, the appropriate syntactic label must be assigned to each attribute in the f–structure.

For each target language in the MT system, we have to define a generation grammar and a generation lexicon. A specification language has been designed for this purpose. Each grammar begins with the specification of grammatical functions (GFs) and atomic attributes susceptible of being synthesised (HGs). Each generation rule is associated with a pair of lists of attributes obtained from the GF and HG sets respectively. The generation rule is triggered if and only if the input f–structure contains those features. Each rule may generate different substructures. For this purpose, we allow the definition of more than one PS rule inside each generation rule, which are discriminated using WHEN conditions. Following the standard LFG–notation, each PS rule is associated with a set of functional equations which call the GENERATE and SYNTHESIS functions.

The SYNTHESIS function takes a triple of syntactic category, semantic root and list of attributes and returns the corresponding lexical item.

3 Specification Languages

3.1 Analysis and Generation Lexicons: Soft–typed Feature Structures (STFS)

This section describes the implementation of specification languages intended for the description of lexical entries in an LFG–like formalism [13].

From the point of view of typification, this language may be placed half–way between untyped formalisms such as PATR–II or DCG and completely typed ones such as ALE or TFS.

3.1.1 Shapes and Multiple Inheritance

The basic object used in STFS is that of "shapes". A shape defines the structure of a lexical entry, allowing for the definition of both typed and untyped structures. STFS also permits the definition of hierarchies of shapes organized as a partial order, including the standard mechanisms of multiple inheritance [15]. Following we show the syntax of the definition of shapes for a verbal entry in the lexicon.

```
DefShapes
    Lex (LU)
    % Auxiliary Shapes
    AGR (agr:(gen,num,per))
    % English verbs
    Everb (CAT:v,LU,MOR,pred,ggf,vtype,aktion,tense,
        asp:(fut,cond,part,prog,nec),
        num,form,
        subj:(role,form,semfeat,@AGR),
        obj:(role,form,semfeat,@AGR),
        {\dots}
        vcomp:(role))
```

3.1.2 Transformational Rules

A shape may have a number of transformational rules associated with it. The application of these rules to lexical entries which match the condition will generate new lexical entries. This is interesting in the case of LFG, since it allows us to define morphological rules and lexical redundancy rules using the same format.

A transformational rule consists of a pattern to be matched and a set of target structures. The pattern filters out the entries to which the rule may be applied and the target structures specify the "effect" of the rule. These consist of a series of new shapes in whose definition we may use values from the original entry and a language for the manipulation of lexical information.

Below we give examples of transformational rules for the generation of –ing forms and examples of a lexical redundancy rule of "intransitivization". This rule generates an intransitive verb entry from a transitive verb which allows optional deletion of its object.

```
% ING RULES
% default, add ``ing''
% look -> looking
TRulePattern (MOR:[EING1])
TRuleTargets {
    (MOR:extract(base->MOR:,[EING1]))
    (LU:strcat(base->pred:,ing),
      asp:(part:pres),num:sing|plur,MOR:null())  }
%
% look up -> looking up
TRulePattern (MOR:[EING1c])
TRuleTargets {
    (MOR:extract(base->MOR:,[EING1c]))
    (LU:strcat(base->pred:[1],ing," ",base->pred:[2+]),
      asp:(part:pres),num:sing|plur,MOR:null())  }
%
% stop -> stopping
TRulePattern (MOR:[EING2])
TRuleTargets {
    (MOR:extract(base->MOR:,[EING2]))
    (LU:strcat(base->pred:,strlast(base->pred:,1),ing),
      asp:(part:pres),num:sing|plur,MOR:null())  }
%
% solve -> solving
TRulePattern (MOR:[EING3])
TRuleTargets {
    (MOR:extract(base->MOR:,[EING3]))
    (LU:strtail(base->pred:,1,ing),
      asp:(part:pres),num:sing|plur,MOR:null()) }
```

```
%
% LEXICAL-REDUNDANCY RULES
TRulePattern (vtype:[opt_obj])
TRuleTargets {
    (vtype:extract(base->vtype:,[opt_obj]))
    (vtype:extract(base->vtype:,[opt_obj]),ggf:[subj,0],obj:null())   }
```

3.1.3 Meta–Relations

Meta–relations are a special type of transformational rule whose goal is the association of a shape with another virtual shape (a meta–shape) from which we have access to the first. In the case of NLP we could create a meta–shape called "lex", which plays the role of a target of the meta–relations of other shapes. Effectively, a query on "lex" will result in a query about any of the shapes associated with it. A direct effect of this strategy is a transparent and compact solution to the problem of lexical ambiguity in NLP. An example is given below. In this case every Everb will be "meta–relationed" with Lex.

```
DefMetas
    MetaPattern Everb ()
    MetaTarget  Lex (LU:base->LU:)
```

3.1.4 Macros and Input Forms

The specification language permits the use of macros, whose behavior is similar to templates in PATR–II, ProFIT, etc.

```
<MP1> = (agr:(gen:masc,num:plural))
```

In the definition of large lexicons it is frequent that many lexical entries vary minimally with respect to each other, sharing many common features. The language permits the definition of Input Forms to ease the lexicographer's job. For example, we could define different input forms using Levin verb classes [10]. Following is an example:

```
DefIForms
    levin_if (pred,MOR,vtype)
        Everb (pred:base->pred,MOR:base->MOR,
        vtype:base->vtype)
```

3.1.5 Basic Entries

Basic lexical entries are associated with shapes. For the definition of an entry, we may take advantage of macros, incorporating multiple inheritance mechanisms.

An entry may also be written making use of Input Forms, to which the corresponding generation models will be applied.

```
ActIForm levin_if
    (bake, [EING3,EED2], [opt_obj])
    (chop, [EING2,EED4], [opt_obj])
```

Once we have defined the entry (either directly or through an input form), transformational rules and meta–relations previously defined will be applied. The result obtained will constitute the knowledge base.

3.2 Analysis Grammar

Grammar rules follow the classical LFG notation. The functional equations associated with each rule are augmented with a control language which allows IF-THEN-ELSE constructions, logical, relational and mathematical operators, string and lists functions such as MEMBER and CONCAT, and specific functions for the control of f–structure wellformedness (COMPLETENESS and COHERENCE).

Any analysis grammar must contain at least the following items:

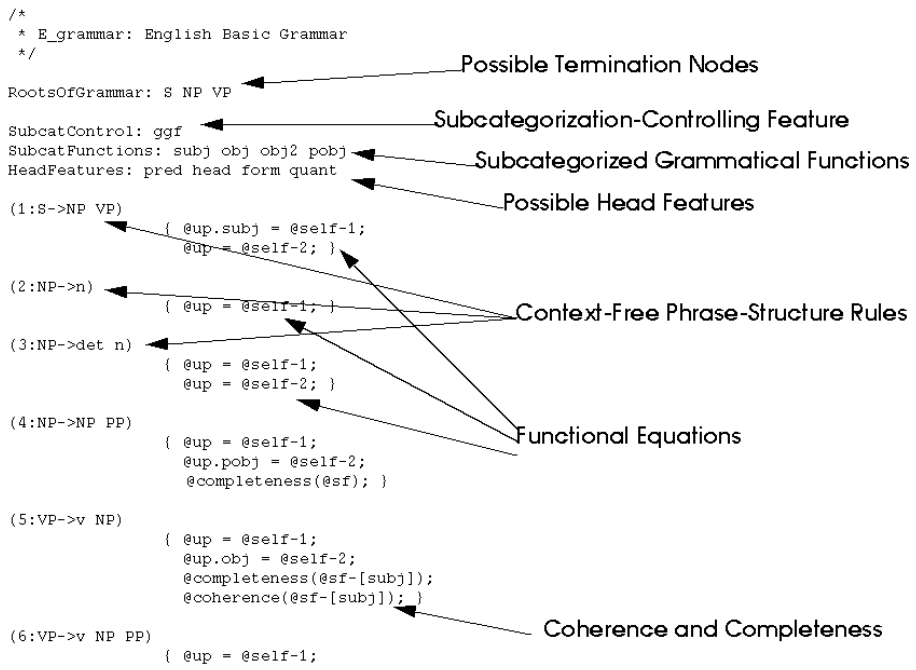

Fig.3. Main Components of Analysis Grammars

3.3 Transfer

The transfer module (from source language f–structure to target language f–structure) allows a uniform definition of structural and lexical transfer rules. Each rule may be associated with conditions and actions. Conditions are triggered according to the input f–structure and actions involve the manipulation of the resulting f–structure by means of specific functions such as NOTRANSFER, TRANSFERAS and overwrite (=c).

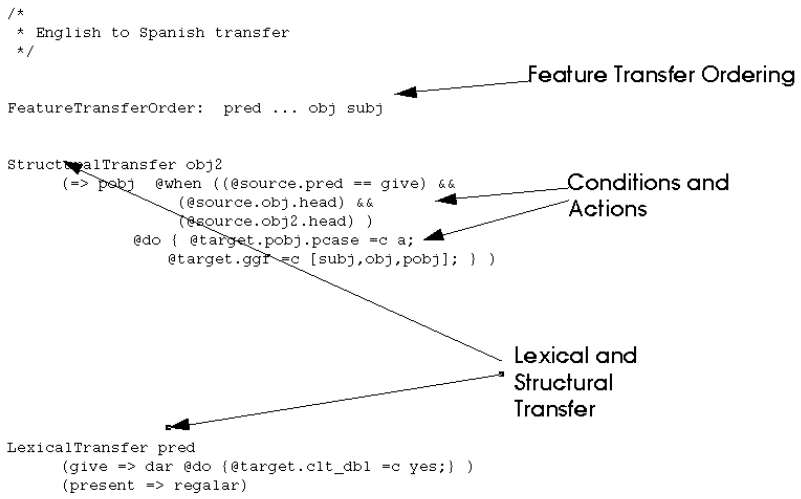

Fig.4. Sample Transfer

3.4 Generation

Generation rules (from target f–structure to target c–structure) assign a c–structure to the input f–structure depending on the attributes present.

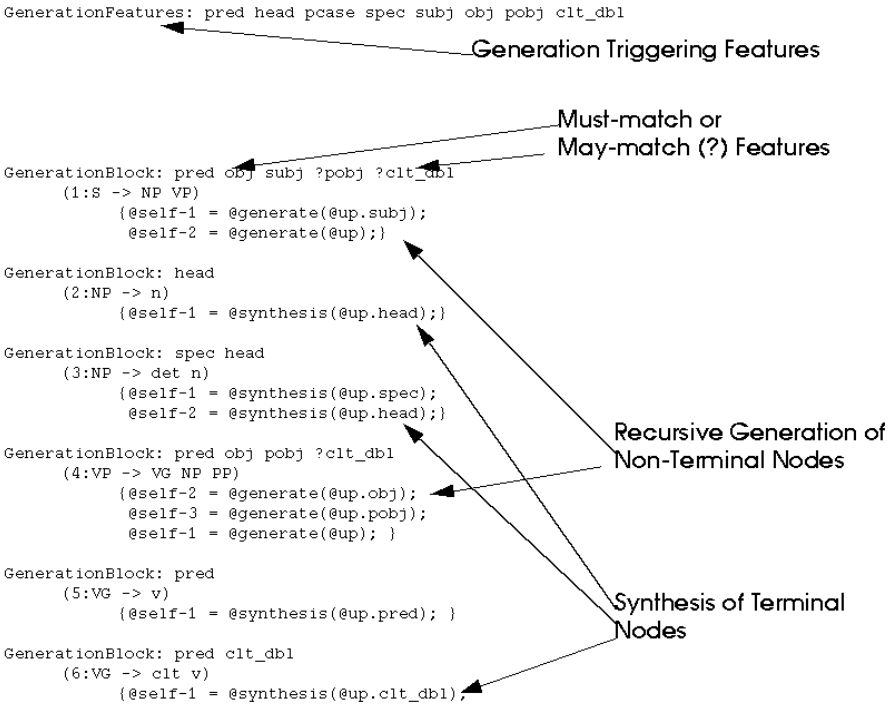

Fig.5. Sample Generation Grammar

References

[1] Alshawi, H. ed. 1991. *The Core Language Engine.* MIT Press.

[2] Alshawi, H., Arnold, D. J., Backofen, R., Carter, D. M., Lindop, J., Netter, K., Tsujii, J. and Uszkoreit, H. 1991. Eurotra 6/1: Rule formalism and virtual machine design study – Final report. Technical report, SRI International, Cambridge.

[3] Carpenter, B. 1992. *The logic of typed feature structures.* Cambridge Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press.

[4] Carpenter, B. and Penn, G. 1994. *ALE. The Attribute Logic Engine Version 2.0.1. User's Guide.* University of Pittsburgh.

[5] Dörre, J. and Dorna, M. 1993. CUF – A formalism for linguistic knowledge representation. In Dörre, J. ed. *Computational Aspects of Constraint–Based Linguistic Description.* Deliverable R1.2.A. DYANA–2. ESPRIT Basic Research Project 6852.

[6] Emele, M. and Zajac, R. 1990. Typed unification grammars. Proceedings of the 13th International Conference on Computational Linguistics, COLING–90. Helsinki.

[7] Emele, M. 1994. TFS – The Typed Feature Structure Representation Formalism. Proceedings of the International Workshop on Sharable Natural Language Resources. Ikoma, Nara, Japan: Nara Institute of Science and Technology.

[8] Erbach, G. 1995. ProFIT: Prolog with Features, Inheritance and Templates. CMP–LG e–print archive: cmp–lg/9502003

[9] Krieger, H. and Schäfer, U. 1994. TDL–a type description language for constraint–based grammars. Proceedings of the 15th International Conference on Computational Linguistics, COLING–94, Kyoto, Japan.

[10] Levin, B. 1993. *Verb classes and alternations. A Preliminary Study.* Chicago: The University of Chicago Press.

[11] Mellish, C. S. 1992. Term–encodable descriptions spaces. In Brough, D. R. ed. *Logic Programming: New Frontiers.* Oxford: Intellect. 189–207.

[12] Quesada, J.F. & G. Amores 1995. A Computational Model for the Efficient Retrieval of Very Large Structure-Based Knowledge Bases. *Proceedings of KRUSE Symposium* University of California at Santa Cruz.

[13] Quesada, J.F. & J.G. Amores (forthcoming 1996). *C for Natural Language Processing.* Studies in Computational Linguistics. London: UCL Press.

[14] Schöter, A. P. 1993. Compiling feature structures into terms: A case study in Prolog. Technical Report RP–55, University of Edinburgh, Centre for Cognitive Science.

[15] Smolka, G., and Aït–Kaci, H. 1990. Inheritance Hierarchies: Semantics and Unification. In Kirchner, C. ed. *Unification.* San Diego, California: Academic Press Inc.

[16] Wah, B. W. ed. 1994. Report on Workshop on High Performance Computing and Communications for Grand Challenge Applications: Computer Vision, Speech and Natural Language Processing, and Artificial Intelligence. IEEE Transactions on Knowledge and Data Engineering, **5** (1), Feb. 1994.