A Producer-Consumer Schema for Machine Translation within the PROLEXICA Project

Patrick Saint-Dizier
IRIT - Université Paul Sabatier
118 route de Narbonne
31062 Toulouse Cedex France
stdizier@irit.fr

Abstract
In this document, we present a dynamic treatment of feature propagation expressed within the Object-Oriented Parallel Logic Programming framework (in Parlog++). This approach avoids a large quantity of useless and linguistically unmotivated lexical data processing and feature percolation. The basic principle of this treatment is based on the idea that a parser for a given language, together with its lexicon and grammar produces intermediate representations which are then comsumed by a generator of a different language. These two processes are viewed as synchronized processes that communicate. The producer sends minimal information and the consumer suspends and asks for more information whenever necessary.

Keywords : lexical feature systems, parsing/generation, Parallel Logic Programming techniques.

1. Introduction

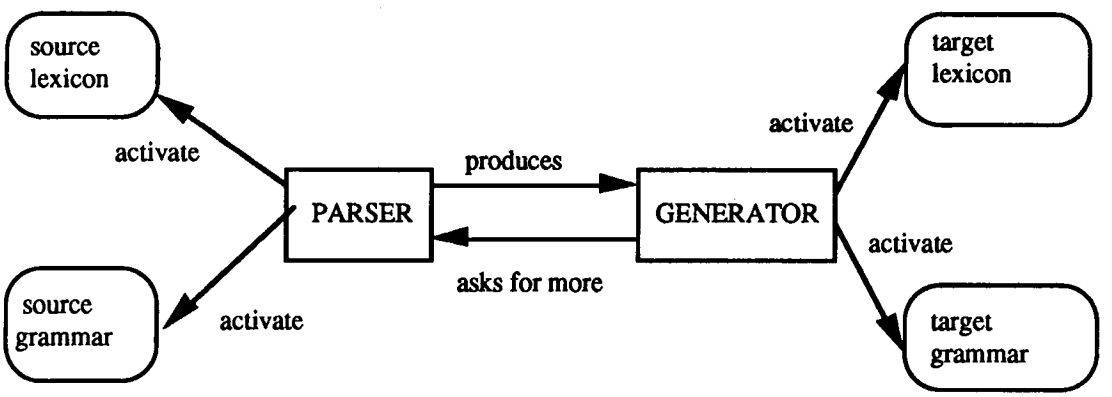
Recent works in Computational Linguistics show the central role played by the lexicon in language processing, and in particular by the lexical semantics component. Lexicons are no longer a mere enumeration of feature-value pairs but tend to have an increasing intelligent behaviour. This is the case, for example, for generative lexicons (Pustejovsky 91) which contain, besides complex feature structures, a number of rules to create new definitions of word-senses such as rules for type coercion. As a result, the size of lexical entries describing word-senses has substantially increased. These lexical entries become very hard to be used directly by a natural language parser or generator because their size and complexity allows a priori little flexibility.

Most natural language systems consider a lexical entry as an indivisible whole which is percolated up in the parse/generation tree. Access to features and feature values at grammar rule level is realized by more or less complex procedures (Shieber 86, Johnson 90, Günthner 88). The complexity of real natural language processing systems makes such an approach very inefficient and not necessarily linguistically adequate. In this document, we propose a dynamic treatment of lexical data and of lexical feature propagation within a machine translation

framework. This treatment is embedded within an Object-Oriented Parallel Logic Programming framework (noted as OOPLP hereafter) which permits an access to feature-value pairs associated to a certain word-sense directly into the lexicon, and only when this feature is explicitly required by the generator, for example to make a control. OOPLP languages such as Parlog (Conlon 89) allow the description of processes that execute different tasks in parallel. Moreover, Parlog allows the specification of synchronization procedures between processes : a process will suspend till another process has produced sufficient information. Parlog++ (Davidson 91), the object-oriented extension of Parlog, combines the power of parallel systems with the structured programming techniques of object-oriented programming. Communication between processes and synchronization is realized in this case by means of messages exchanged between objects.

A MT system can then be viewed as a set of producer-consumer pairs. The central pair is composed of (1) the parser which produces fragments of intermediate representations from a sentence of the input language and (2) the generator which consumes these representations to produce a sentence in the target language. The parser is tuned to send minimal information and the generator can ask the parser to produce more in cases when it is required. Other elements such as lexicons and grammars can also behave as producers with respect to their associated parsers or generators.

Lexicons and grammars are activated when information is required, the parser is always active when there is a sentence to parse. When it is asked for more information by the generator, it suspends its current work to produce the required information. Finally, the generator suspends till it has sufficient information to go on working. This global schema can be represented as follows :



As shall be seen more in detail from an architectural point of view, the lexcions and the grammars can have various forms. For the sake of understandability, let us say that the genarl form of these components is based on the notion of unification grammar. They contain usual lexical and grammatical information.

The motivations for this approach are twofold. The first motivation is obviously efficiency. When translating a sentence, only small portions of the lexical entries corresponding to the words of the surface sentence being processed are used. It is preferable to delay this global

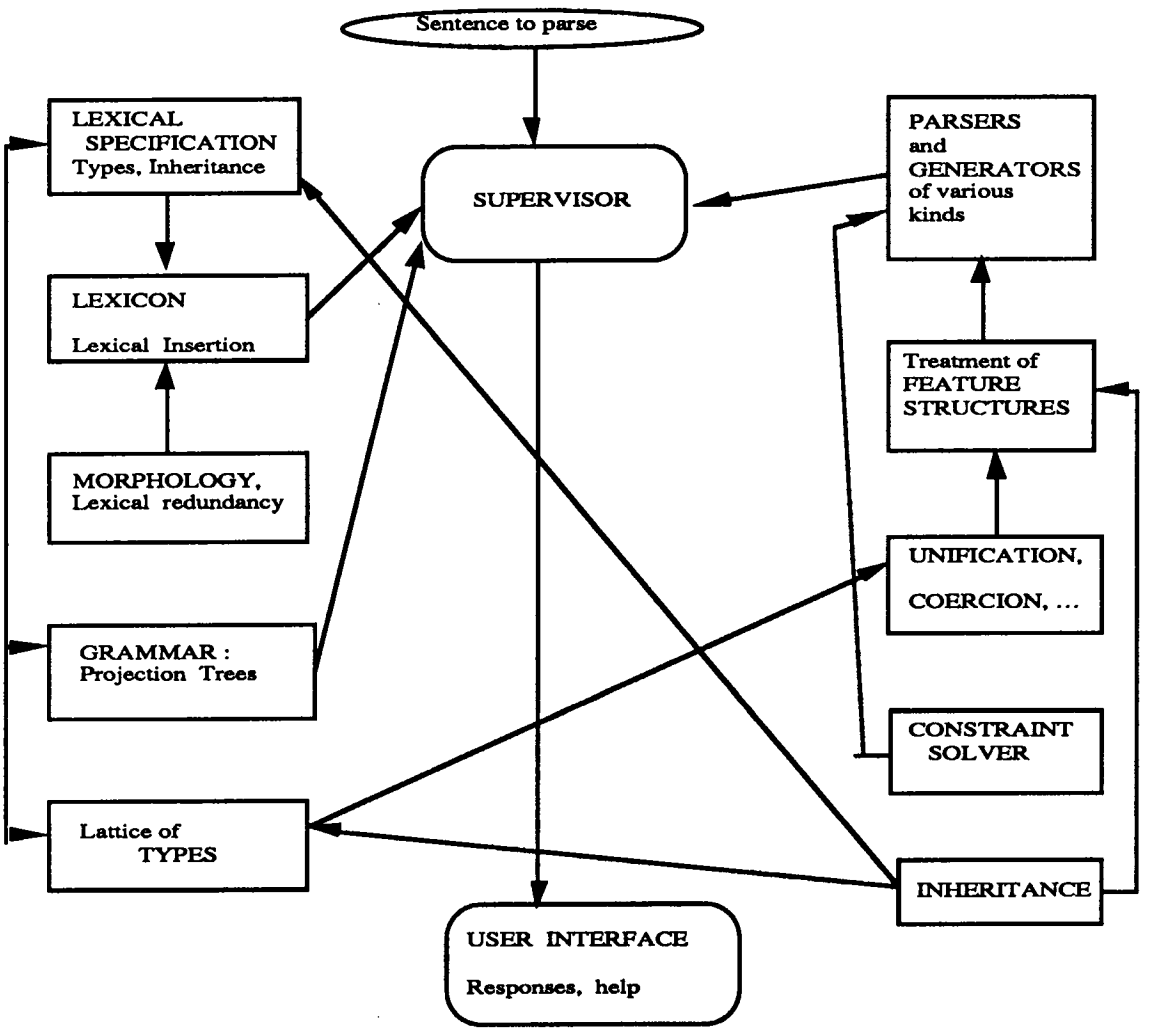
percolation and only to extract the relevant information when required. The second motivation to our approach is linguistic adequacy. Most of the information conveyed by features is often linguistically relevant (and thus used) very locally in a parsing/generation tree. For example, in:

John opened the door.

the aspectual value of the verb to open is only relevant at the level of the VP category, i.e. the level of the maximal projection of the category. There is no reason to have a more or less specified aspectual feature at lower levels, e.g. V^1 and V^0 in the X-bar system.

2. The project PROLEXICA

The project PROLEXICA we are currently developing as a testbed for studying different processing strategies, computer models as well as linguistic models, is based on the notion of lexical projection. Lexical entries are relatively comprehensive and each lexical entry is associated to a set of basic trees roughly corresponding to the maximal projection of the word associated to that lexical entry. This approach must not be confused with TAGs, since we do not have predefined trees that go beyond any maximal projection. The overall architecture of Prolexica is the following :



The generation component also includes a lexical choice component, not shown on this diagramme. In a MT system, these two processes communicate by means of a variety of inter-lingua representations. A transfer-based approach could also be used. As can be seen, the system uses a priori the same lexicon and grammar for both the generation and the parsing of a sentence. Similarly, processes such as unification, inheritance, coercion and constraint resolution are used in both systems.

3. Parallel Logic Programming and the Producer-Consumer Schema

Parallel languages such as Parlog allow programmers to express synchronization between processes. They also allow the specification in a simple way of message exchanges between these processes. A message can be, for example, a data structure which is only partially instantiated at the beginning of a process and which is filled in step by step by the producer, upon requests from the consumer. Parlog supports full and-parallelism and a committed-choice or-parallelism with guarded Horn clauses. In other terms, the litterals in the body of a clause are executed in parallel and the different clauses corresponding to a definition are also considered in parallel. A commitment is made on the clause for which unification with the head and execution of the guard suceeds.

The producer-consumer schema that we consider goes beyond the usual competition between processes. We present here a synchronization technique which permits the suspension of a process till another process has completed his work, or has done a minimal part of it that permits the suspended process to resume working. These processes can be viewed as independent machines which communicate by means of messages. These messages can moreover be considered as objects. It should be noted that it is only the content of these messages which determines the suspension of a process since suspension is mainly provoqued by the fact that an input mode variable has not yet been instantiated.

OOPLP is of much interest for concurent logic programming. Notions of encapsulation of data and programmes, state changing and stream manipulation allow the removal of some of the weaknesses of concurent logic programming where the only structure is the predicate.

Concurent logic programming is also of much interest to OOPLP. Concurent logic programming makes available simple ways to express synchronization and concurency within and between objects. Most concurent logic programming also offer committed choice non-determinism, which is often the most appropriate strategy for OOPLP.

We now present the features of Parlog++ (Davidson 91). Concurent logic programming and OOLP systems usually have the five following characteristics:

- an object is viewed as a process which calls itself recursively (to maintain it 'active') and which has an internal state stored in unshared variables,
- communication between objects is based on the instantiation of variables in messages,
- an object becomes active when it receives an appropriate message, otherwise it is suspended,
- an object instance is created by process reduction,

- a response to a message is characterized by the binding of a shared variable in a message.

In Parlog++, the basic idea is that an object is viewed as a process which suspends till it gets a message to process. This can roughly be sumarized by the following Parlog programme, where obj is an abstract object:

```
mode obj( ?, ? ).    % obj has 2 input mode arguments
obj( [Input_message| Next], Current_state) <-
  action1_obj(Input_message, Current_state, New_state) :
  obj(Next, New_state).
obj( [Input_message| Next], Current_state) <-
  action2_obj(Input_message, Current_state, New_state) :
  obj(Next, New_state).
etc...
obj( [ ], [ ] ).
```

The object obj has one input message stream, its first argument, and one state variable, the second argument. That state variable may originate a message which will be executed by another object in the current programme. We have mentioned two different possible actions, that will be triggered depending on the input message. This abstract example shows that an object can (1) try in parallel different actions which could potentially be triggered from a given message and (2) process in parallel and simultaneously different messages.

Let us now introduce the main features of the language Parlog++. A Parlog++ class can be summarized as follows, in a kind of BNF form:

```
< class name > .
< variable declarations >
{ initial  <actions> }
clauses
   < clauses >
{ code
   < predicates > }
end.
```

Sections between curly brackets are optional. Initial, clauses, code and end are reserved keywords. A Parlog++ class begins with a name and then the variables of the class that define streams and states are declared, if any. These variables, as shall be seen later, can be made visible or invisible to the user (i.e. the user will or will not have access to their contents). The optional initial section contains clauses which must be executed before the clauses section. The clauses section recieves and produces streams corresponding to messages. It also contains the clauses that treat these messages. The general form of a clause in that section is the following:

```
< input message > => < actions to execute > <action separator>
```

The set of < actions to execute >, is a Parlog clause body, possibly containing Parlog++ operations, while <action separator> is either the symbol '.' to allow OR-parallel search between the different clauses or the symbol ';' to realize a sequential search. The code section contains portions of code used in the clauses of the clause section. This code section is private

to the object. This can be very useful to structure programmes. Finally, the class ends by the symbol end.

A clause is activated by unifying the input message with its message part, situated before the symbol =>. If the unification succeeds and if the guard is true (if any), then the clause body is executed. Clauses may have their own local variables.

Let us consider a very simple example of an object that represents a lexicon:

```
lex1.
clauses
  last => end.
  lex(the, Cat) => Cat = det.
  lex(a, Cat) => Cat = det.
  lex(book, Cat) => Cat = noun.
  lex(has, Cat) => Cat = verb.
  lex(pages,Cat) => Cat = noun.
end.
```

Input messages are of the form:

```
lex( <word >, Cat )
```

and the object returns a value for Cat, if the word is in the object, e.g.:

```
lex(book,C).
C = noun.
```

4. Translating sentences within a producer-consumer schema

Let us now examine by means of a simple example how the producer-consumer schema works and show its advantages. In order to avoid useless computations and propagation of feature values, the exchanges between the parser and the generator are kept minimal. The degree of minimality can be parametrized depending on the source and target languages. For example, in languages of the same family it can sometimes be sufficient to produce a syntactic tree as an intermediate representation. However, in most situations, it is necessary to produce a semantic representation such as an argument structure-based or an interlingua representation (Dorr 93). The examples below will make use of a very simple intermediate representation based on the argument structure-based representation currently produced in Prolexica. This representation remains superficial but may be deepened and, for example, an LCS-based (Jackendoff 90) representation may be required for some complex sentences or phrases.

Let us now consider the translation of :

Jean aime manger into *Johan hat essen gern. (or Johan esst gern)*

'John likes to eat'

The full semantic representation of the input French sentence is the following in Prolexica :

```
rept([prop(eventuality(
            predicate([aimer]), negation(no_neg),
            arguments([arg(role(agent),
                quantifier([empty]), dom_restr([jean])),
              arg(role(theme),
```

```
                    quantifier([empty]),dom_restr([manger]))]),
          modality(no)), sentence_negation(no),
          sentence_modalities(time(no),place(no),manner(no)))]  )).
```

The parser (i.e. the producer) makes a bottom-up parse from left to right. It therefore produces first the representation of the subject argument :

```
arg(role(agent), quantifier([empty]), dom_restr([jean]))
```

which is included into the sentence semantic representation. At this stage, the remainder of the representation remains empty. The generator has enough information to start producing a noun phrase which will become a subject or an object depending on the grammar of the target language.

The generator is based on a technique presented in (Saint-Dizier 91). It is basically bottom-up and it is based on the notion of generation point. The basic formalism is the following :

```
generate(<fragment of semantic representation>, Generated_Phrase) :-
  <call to a set of specific grammar rules associated to the representation>,
  <call to the lexical selection module>.
```

These two calls may be executed in parallel and may cooperate. Besides this semantic representation, these two calls may need additional information about e.g. aspect, semantic selectional restrictions, deeper semantic representations (e.g. LCS), etc. The absence of this information, which is declared as input mode variables or data-structures entails the suspension of the generator and the production of a message to the parser which will then search in the lexical data to get the information.

This case occurs with the verb 'aimer' in French. The parser produces the predicate 'aimer' which is not ambiguous in French. In German, it may be translated into either 'lieben' or 'gern haben' (or gern + infinitive verb). At that level, the generator cannot make any decisions by just consulting the target language grammar and lexicon. It thus suspends and asks for another semantic representation, for example an LCS-based representation of the form :

$$
\begin{array}{l}
([_{State}\ \text{BE} \\
\ ([_{Thing}\ \text{Jean}\ ], \\
\ \ [_{Event}\ \text{EAT}\ ] \\
\ \ [_{Manner}\ \text{LIKINGLY}\ ])).
\end{array}
$$

The system roughly works as follows. The LCS-based semantic representation is the semantic 'entry point' of the lexical entry corresponding to the verb 'gern haben'; it allows a priori a non-ambiguous selection of that entry. It is declared as an input mode variable. If it is not instantiated in the call, this variable together with its associated feature name triggers the production of a standard message which is sent to the parser. The parser has a routine that recognizes the contents of the message and triggers the production of that representation, which is then sent again to the generator.

Finally, the case of the translation of 'manger' into either 'essen' or 'fressen' is solved in a different way and illustrates another aspect of the propagation technique. The basic principle is to limit as much as possible the percolation of feature-value pairs in a generation system. If there were no ambiguity in the translation, the translation would have been done directly. In our example, the generator has to ask the target lexicon about the semantic feature of the

subject. The target language lexical system is triggered and it produces the value 'human' (possibly computed by means of an inheritance mechanism internal to the lexicon). The same strategy is used : the semantic type of the subject is declared as an input mode variable for both essen and fressen.

As can be seen, apart from basic information which are absolutely necessary, the other sources of information are triggered only when required by the generation system. In a certain sense, we can say that it is the generator which pilots the translation system.

5. Machine Translation in Parlog++

In this section, we present two examples of a producer-consumer schema between a source language and a target language processor. As explained above these two systems are synchonized and exchange messages. They thus minimize the number of exchanges to those which are absolutely necessary to the system. Of particular importance are those exchanges from the lexicon, which can be really very important.

5.1 Accessing to semantic feature values

Let us first treat the relatively simple case of *essen* versus *fressen* as possible German translations of the verb *manger* in French. Here is a simple portion of an object that processes sentences. The rule presented here under the message s(X,Y,T) processes intransitive verb constructions of the form :

sentence --> proper_noun, intransitive_verb.

X and Y represent the difference lists as in DCGs. T is the resulting semantic representation, which has here the following form:

pred([<predicate name>, <predicate argument>]).

The programme is the following. For the sake of readability, the lexicon has been incorporated into this object :

```
strans.
clauses
 last => true.
 s(X,Y,T) =>
            word(X,Z,F,proper_noun,W1),
            word(Z,S,F1,verb,W2),
            Y = S,
            T  =  pred([W2,W1]).
 semantics(W,Sem,Cat) =>
            word([W|_],_,feat(_,_,SSem),Cat,_),
            Sem = SSem.
code
 mode word(?,^,^,?,^). % declaration of arguments : ? is input mode, ^ is output mode
  word([jean|X],X,feat(masc,sing,human),
     proper_noun,jean).
  word([marie|X],X,feat(fem,sing,human),
```

```
      proper_noun,marie).
  word([lola|X],X,feat(fem,sing,animal),
      proper_noun,lola).
  word([mange|X],X,feat(sing,_),verb,manger).
  word([marche|X],X,feat(sing,_),verb,marcher).
end.
```

The message semantics is a utility which allows any other object to have access to the semantic feature(s) of the word W of category Cat. Clearly, such a message will be produced by the target language system in order to get, whenever required, and only in that case, the necessary data. The target language generator system is the following:

```
starg.
invisible To ostream
initial  strans(To).
clauses
 last => true.
 s(pred([W1,W2]),X)  =>
    not( W1 = manger) :        % end of guard, realizes commitment
     word([A|_],_,feat(_,SSem),verb,W1),
     X = [W2, A].
 s(pred([W1,W2]),X)  =>
      W1 = manger :
     To :: semantics(W2,SSem,proper_noun),        % call to the object strans
     word([A|_],_,feat(_,SSem),verb,W1),
     X = [W2, A].
code
 mode  word(?,^,^,?,^).
  word([esse|X],X,feat(sing,human),verb,manger).
  word([fresse|X],X,feat(sing,animal),verb,manger).
  word([wandert|X],X,feat(sing,animal),verb,marcher).
end.
```

This example shows how specific cases, treated in parallel with more general cases, are identified. The guard permits to postpone the committed choice of the parallel system till this guard is evaluated to true, avoiding thus incorrect choices. In the case of W1 = manger, then the generator suspends till it gets the semantic feature value of the subject argument. This is realized by the call:

```
To :: semantics(W2,SSem,proper_noun)
```

which is sent to the strans object and which is executed in parallel with other activities that object may have. During that execution, the starg object has suspended this activity. It resumes working when it gets the information back.

5.2 Cooperation between source and target processing modules

Let us now present in more generality the cooperation between the source language and the target language processors. The source language processor contains calls to the target language generator, that allows the generator to start working as early as possible. The lexicon has the same format as in section 5.1, it is thus ommitted here for the sake of readability. Here is a schema of of the target laguage processor, ommitting variable declarations to facilitate readability :

```
superv_french.
        % call samples, as illustrated in section 5.1
semantics(Sem, np) => To :: semantics(Sem, np).   % redirected to the np
semantics(Sem, vp) => To :: semantics(Sem, vp).   % redirected to the vp
end.
sentence_fr.    % object treating sentences
s(X,Y,T) =>
      % X and Y are difference lists, T is an internal representation of the sentence
            To :: np(X,Z,T1),   %  call to the NP object in npfr
            To1 :: vp(Z,Y,T2),  % call to the VP object in vpfr
            T = f(T1,T2),          % T is a certain composition function f of T1 and T2
            Toeng :: s(X1,Y1,T).  % call to the English supervisor
end.
npfr.    % object treating noun phrases
np(X,Y,T) =>  To :: det(X,Z,T1),
              To1 :: n(Z,T,T2),    % calls to the French lexicon
              T  =  f(T1,T2),
              Toeng :: np(X1,Y1,T).  % call to the English supervisor
etc...
end.
vpfr.    % object treating verb phrases
vp(X,Y,T) =>  To :: v(X,Z,T1),
              To1 :: np(Z,T,T2),    %  call to the NP object in npfr
              T  =  f(T1,T2),
              Toeng :: vp(X1,Y1,T).  % call to the English supervisor
etc...
end.
```

All calls from the source language processor are directed to the target language generator, to guarantee a better independence between the two systems. As sson as the source language processor has processed a portion of the sentence to translate, it sends it to the target language processor which starts processing it. In case it needs more information (see section 5.1 above), it send suspends and sends a request to the source language processor, to get that information. Here is the general schema of the target language processor:

```
superv_eng.
 % supervises all the actions done by the English modules
np(X1,Y1,T) => To1 :: np(X1,Y1,T).    % redirected to the npeng module
vp(X1,Y1,T) => To2 :: vp(X1,Y1,T).   % redirected to the vpeng module
s(X1,Y1,T) => To3 :: s(X1,Y1,T).       % redirected to the sentence_eng module
end.
sentence_eng.
s(X,Y,T) =>   To :: np(X,Z,T1),   %  call to the NP object
              To1 :: vp(Z,Y,T2),  % call to the VP object
              T  =  f(T1,T2).
end.
npeng.
np(X,Y,T) =>  To :: det(X,Z,T1),
               To1 :: n(Z,T,T2),
               T  =  f(T1,T2).
etc...
end.
vpeng.
vp(X,Y,T) =>   To :: v(X,Z,T1),
              % similarly to section 5.1, may contain calls to semantics
               To1 :: np(Z,T,T2),
               T  =  f(T1,T2).
etc...
end.
```

Conclusion

In this short text, we have shown how the access and the processing of lexical data in a machine translation system can be used in a dynamic way so as to avoid any useless computations. We have given a few examples that show how this general technique can be realized within an object oriented parallel logic programming framework. The combination of object oriented techniques and parallel programming produces a very interesting framework, of a good degree of expressivity and completeness. Examples have been given in Parlog++. Because Parlog++ is still in a stage of prototyping on small machines, we do not have yet an efficient system, however, the approach shows the advantages of the method.

References

Aït-Kaçi, H., Nasr, R., LOGIN: A Logic Programming Language with Built-in Inheritance, *journal of Logic Programming*, vol. 3, pp 185-215, 1986.

Chomsky, N., *Barriers*, Linguistic Inquiry monograph nb. 13, MIT Press 1986.

Colmerauer, A., An Introduction to Prolog III, *CACM* 33-7, 1990.

Conlon, T., *Programming in Parlog*, Addison Wesley, 1989.

Davidson, A., Parlog++ : A Parlog Object Oriented Language, PLP research note, UK, 1991.

Dincbas M., Van Hentenryck P., Simonis H., Aggoun A., Graf T. and Berthier F. (1988) "The Constraint Logic Programming Language CHIP," *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp 693-702, ICOT, Tokyo.

Dorr, B., *Machine Translation : A View from the Lexicon*, MIT Press, to appear 1993.

Emele, M., Zajac, R., Typed Unification Grammars, in proc. *COLING'90*, Helsinki, 1990.

Freuder E. C. (1978) "Synthetising Constraint Expressions," *Communications of the ACM*, 21:958-966.

Gazdar, G., Klein, E., Pullum, G.K., Sag, I., *Generalized Phrase Structure Grammar*, Harvard University Press, 1985.

Günthner, F., Features and Values, Research Report Univ of Tübingen, SNS 88-40, 1988.

Jaffar, J., Lassez, J.L., Constraint Logic Programming, *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987.

Jackendoff, R., *Semantic Structures*, MIT Press, 1990.

Mackworth A. K. (1987) "Constraint Satisfaction," in Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, Wiley-Interscience Publication, New-York.

Pollard, C., Sag, I., *Information-based Syntax and Semantics*, vol. 1, CSLI lecture notes no.13, 1987.

Pustejovsky, J., The Generative Lexicon, *Computational Linguistics*, 1991.

Saint-Dizier, P., Processing Language with Types and Active Constraints, in proc. *E. ACL 91*, Berlin, 1991.

Saint-Dizier, P., Generating sentences with Active Constraints, in proc. *3rd European workshop on language generation*, Austria, 1991.

Sheiber, S., An Introduction to Unification-Based Approaches to Grammar, *CSLI lecture notes no 4*, Chicago University Press, 1986.

Van Hentenrick P. *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, 1989