SCSL : a linguistic specification language for MT

paper presented for
COLING-86
Bonn, 25-29 August 1986

by

Rémi ZAJAC

GETA, BP 68
Université de Grenoble
38402 Saint-Martin-d'Hères, FRANCE

ABSTRACT Nowadays, MT systems grow to such a size that a first specification step is necessary if we want to be able to master their developement and maintenance, for the software part as well for the linguistic part ("lingwares").

Advocating for a clean separation between linguistic tasks and programming tasks, we first introduce a specification/ implementation/ validation framework for NLP then SCSL, a language for the specification of analysis and generation modules.

KEY-WORDS Machine Translation, Natural Language Processing, Specification Language.

I. INTRODUCTION

In most NLP and second generation MT systems, the information computed during the process is generally represented as abstract trees, very common description tools used in linguistics. The modules implementing the various steps are written in Specialized Languages for Linguistic Programming (SLLP) (see for example <Vauquois85a>, <Nakamura84>, <Slocum84>, <Maas84>).

In spite of the expressive power of SLLP compared to traditionnal programming languages such as LISP, the conception and the maintenance of programs become more and more difficult as the complexity of "lingwares" grows.

To take up this challenge, we introduce in the field of computational linguistics the specification/ implementation/ validation framework which has been proved valuable in traditionnal programming. This leads to the intoduction of new tools and new working methods. The expected benefits for computational linguists are allowing them to facilitate the conception of the linguistic parts of NLP systems, to increase the speed of realisation, to improve the reliability of the final system and to facilitate the maintenance.

Writing an analysis program with a SLLP, the computational linguist must define the set of strings to be analysed, the structural descriptor corresponding to an input string, the strategies used for the computation of the descriptor, the heuristics used for ambiguity choices and the treatment of wrong inputs (errors). He generally writes a more or less precise and comprehensive document on those problems and begin programming from scratch. This method is highly unfeasible with large lingwares. We advocate for the use of a more stringent methodology which consist of :

1. Specify formally (i.e. using a formal language) the valid inputs and the corresponding outputs : the specification must be comprehensive and neutral with respect to the choices of implementation. At this stage, the computational linguist is concerned only with linguistic problems, not with programming. An interpreter for the specification language should be used to write and debug the specification.

2. Specify the implementation choices for data structures and control (decompostion into modules, strategies and heuristics) and the treatement of errors. This specification depends on the input/output specification and may partially depend on the kind of SLLP to be used for implementation. It should be as formal as possible, at least a strictly normalized document.

3. Implement the module specified using a particular SLLP.

4. Validate the implementation : the interpreter of the specification language should be used to prepare a set of valid inputs/outputs; the results of the execution of the module to be validated on the input set is compared to the output set.

An integrated software environement offering the developement tools and insuring the coherence between the developement steps should be provided to facilitate the use of the methodology.

As a first step toward this direction, we introduce a linguistic specification language for which an interpreter is being implemented. Those tools are used in the first and fourth steps as defined below and are being integrated in a specialized environement based on the specification language <Yan86>.

II. LINGUISTIC SPECIFICATION

1. A SPECIFICATION FORMALISM

Before presenting the specification language itself, we shall consider what properties that such a language should have.

Most problems in NLP systems are found in the analysis stage (and some in the transfer stage in MT systems). The major gain should be to clarify the analysis stage using the proposed framework. Thus, a linguistic specification language should :

- define the set of valid input strings;
- define the corresponding ouputs (structural descriptors of strings);
- define the mapping between those two sets.

Analysis and synthesis are two complementary views of a language defined by a formal grammar. We should reasonably expect that a linguistic specification language should be equally used for the specification of analysis and synthesis modules <Kay84>.

Formal grammars define formal languages, and formal grammars does not make any reference to the situation (the global context in which sentences are produced), thus formal languages used to describe natural language sub-sets must allow the expression of ambiguities and paraphrases. An element of the mapping should be a couple (string, tree) where many trees are generally associated to one string and conversely, many strings are associated to one tree.

The advantage of modularity is admitted and the description of the mapping should be done piece by piece, each piece describing a partial mapping and the total

mapping is then obtained by the composition of partial mappings (e.g. unification as in FUGs <Kay84>).

An important feature of such a language is that a linguistic specification should be written by linguists who have no a priori knowledge in computer science : a linguist must be able to concentrate only on liguistic problems and not on computer science problems. The formalism should be clean of all computer science impurities, the mechanism of composition should be clear and simple.

Within this framework, a graphic formalism for the specification of procedural analysis or generation grammars, the "Static Grammars" (SG) formalism has been developped at GETA under the direction of Pr.B.Vauquois <Vauquois85b>. This formalism is now used in the French MT National Project to specify the grammars of an industrial English-French system. Up to now, SGs were hand-written and cannot be edited on computer because of the use of graphs. This formalism has been modified in order to realize a software environement based on SG (structural editor, interpreter, graphic outputs, ...). It is called "Structural Correspondance Specification Language" (SCSL). A grammar written in SCSL is called "Structural Correspondance Specification Grammar" (SCSG).

SCSL (sect.III) allows one to write the grammar of any interesting formal language such as programming languages or sub-sets of natural languages. This formalism is quite general and does not depend on a particular linguistic theory. GETA, under the direction of Pr.B.Vauquois, has elaborated its own linguistic framework and methodology from which this work directly descends, but it is nevertheless perfectly possible to write grammars within different linguistic frameworks. We emphasize this point because the distinction between the formalism properties and the linguistic theory properties is not always clear. Moreover, it may be tempting to wire the properties of some linguistic theory within a particular formalism, and this is sometimes done, leading to confusion.

2. IMPLEMENTATION AND VALIDATION OF LINGUISTIC MODULES

As mentioned earlier, a SCSG is used for the specification of analysis or generation modules written in one of the SLLP of the ARIANE system. Defining a mapping, a SCSG is neutral with respect to implementation choices which are essentially algorithmic in nature (organisation in modules, control, etc) and with respect to intrinsic ambiguity choices which are essentially heuristic in nature.

The same SCSG may be used to specify the inputs/ouputs of different procedural grammars, each of which implementing different strategies and heuristics for comparative purposes : the result must nevertheless correspond to the same specification.

The interpreter (not yet fully implemented) is used for debugging a SCSG (tests, traces, ...) and for the empirical validation of procedural grammars for analysis or generation: the function computed by a procedural grammar must be included in the mapping defined by the SCSG specifiying the procedural grammar.

The interpreter may compute the trees corresponding to an input string (analysis) or the strings corresponding to an input tree (generation). A chart identifier may define an entry point for the interpreter.

Before an execution, one can type in different trace commands. At the end of an execution, the trace and the derivation may be printed.

One can trace for different charts (step-by-step or otherwise) a tentative application of a chart, a sucess, a failure or a combination of these parameters. In the step-by-step mode, the interpreter stops on each traced trial/sucess/failure and it is possible to type in new commands (trace, untrace, stop) and chose the next chart to be applied.

An output trace element have the following general pattern (several levels of details are avaible) :

<chart_id>, <tree_occurence>, <string_occurence>.

III. THE LANGUAGE

To give a flavour of the specification language, we introduce a simplified version. Unnecessary (but essential for practical use) constructs of the language are removed. A more abstract view has been studied in <Zaharin86>.

A SCSG describe simultaneously :

- the set of strings of the language;
- the set of structural descriptors of the language;
- the mapping between those two sets.

A SCSG is composed of "charts". The mapping between the string language and the tree language is decribed in parts : a chart decribes a partial mapping (set of valid sub-strings <-> set of valid sub-trees), the total mapping is obtained by the composition of partial mappings (sect.IV).

SCSL is a language using key-words : every important syntactic unit begins with a key-word (e.g. CHART). Identifiers begin with at least one lotter, designators begins with at least one digit. Designators are preceded by a prefix indicating their type.

A SCSG begins with the declaration of labels and decorations, and then followed by the charts. Charts consist of a tree part and a forest part describing respectively a tree pattern and a forest pattern. We then have the contexts part and lastly the constraints part (sect.III.2).

SCSL do not have the concept of assignement : a chart defines correspondance between a tree and a forest constrained by a boolean expression on the patterns of the chart.

The basic construct of the language is a labeled and decorated tree pattern : each node of the described trees is a couple (label, decoration). The label have the string basic type, the decoration have a hierarchical definition which use the SCALAR and SET constructors. A constraint is a boolean expression on the labels and decorations of the nodes of the patterns.

1. LABEL, DECORATION AND TREE PATTERNS

Most of SLLP use trees as basic data structure. Some associate to a tree or to a node attributes, essentially a set of variable/value pairs which may be manipulated with a few operators. To offer a more powerful description tool, a SCSL node tree is a couple (label, decoration) where the decoration is a hierarchical attribute structure. This is intended to facilitate the manipulation of complex sets of attributes through a unified view.

1.1. Label

The label is traditionally a non-terminal of a grammar, but it may be viewed as a particular attribute of a tree. The type definition of labels is expressed with a regular expression. The operation on this type is equality.

Exemple :

```
LABEL lbl = ( "b".("a")* )+   + "S" + "A" + "B"
```

1.2. Decoration

The decoration is interpreted as an oriented non-ordered tree where attribute identifiers (SCALAR or SET type) are the labels of the nodes and the values of the attributes are the forests that they dominate (in the

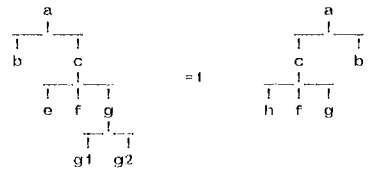
actual version of SCSL, attributes may have the STRING or INTEGER types with the associated operators).

For the SCALAR type, the operation is equality. For the SET type, the operations are union, intersection and set difference. Relational operators are equality, menbership and inclusion.

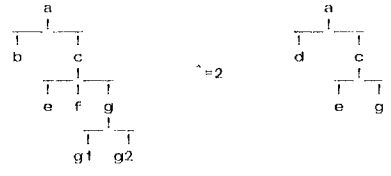
The operations are defined on a hierarchical set structure : one must indicate on which level an operation is defined by suffixing the operator with an integer. The default value is the first level; "*" is used for the deepest level.

Examples :

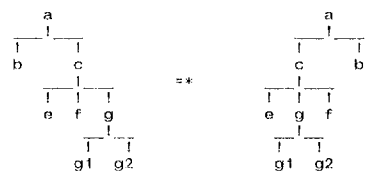
a(b,c(e,f,g(g1,g2))) = a(c(h,f,g),b) is true :



a(b,c(e,f,g(g1,g2))) =2 a(d,c(e,g)) is false :



a(b,c(e,f,g(g1,g2))) =* a(c(e,g(g1,g2),f),b) is true :



Toy example of decoration for noun phrases :

```
DECORATION  deco : SET (

  semantic_relation  : SCALAR (
          instrument, quantifier, qualifier),
  syntactic_function : SCALAR (
          coordination, governor, subject),
  category : SCALAR (
          noun : SCALAR (
              semantic : SET (animate, measure)),
          adjective : SCALAR (
              ordinal, cardinal, noun_phrase_quantifier)
          determiner : SCALAR (quantifier),
          subordinator : SCALAR (preposition) )
```

1.3. Tree pattern

The basic notion of the language is a labeled and decorated tree. The types of a node, a tree, a forest are defined by the declaration of the labels and the decorations.

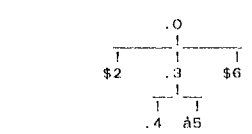
A chart should be a comprehensive description of a linguistic fact which may have different realisations : the decoration allow the manipulation of sets of attributes at different levels of detail, the structure should describe a whole family of trees.

The structure of a tree pattern is described with designators which are implicitly declared. The scope of a designator is reduced to a chart. A designator begins with one digit.

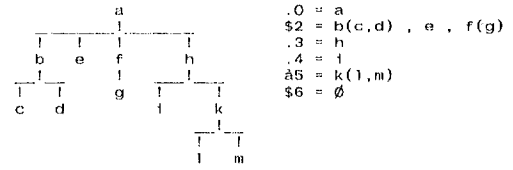
- a node designator is prefixed with ".". The content of a node is accessed by means of decoration and label identifiers : the label of a node .1 is accessed by lbl(.1) (if the label is declared as "lbl"), its decoration by deco(.1).
- a tree designator is prefixed with "À". The tree may be reduced to a single node.
- a forest designator is prefixed with "$". The forest may be empty.

A tree pattern describe a set of trees, each tree being completely describe in width and depth.

Example : the pattern .0($2, .3(.4, $5), $6) :



may be instantiated by :



Here, labels are for a couple (label, decoration).

2. CHARTS

A chart has the following pattern :

```
CHART <chart_id>

 <TREE>              -- tree pattern,
 <FOREST>            -- sequence of tree patterns,

 <LEFT CONTEXT>      -- set of cuts of the derivation tree
 <RIGHT CONTEXT>     -- containing the tree pattern.

 <CONSTRAINTS>       -- boolean expression on labels and
                     -- decorations.
```

2.1. Tree and forest parts

The tree part describes a set of trees with the following syntax :

```
TREE   <tree_pattern>
```

The forest part describes a set of sub-strings with the following syntax :

```
FOREST   <forest_pattern>
```

The element of the forest pattern may be :

- a string element described directly;
- a sub-string described indirectly using the corresponding structure (tree), defined by some chart.

The forest pattern is a sequence of tree patterns described by a regular-like notation : a tree pattern suffixed by "+" may be iterated, by "?" optional and by "*" optional or iterated. Contrary to regular expressions, one can use these notations for single tree patterns only.

To have simpler notations, an iterated tree pattern, e.g. (.1(.2,.3))*, will be written .1*(.2,.3) and the same convention will hold for "?" and "+". Such a pattern must be used as a whole and is interpreted as a list : a boolean expression on nodes of such a pattern is interpreted as an expression on the nodes of each tree of the list.

Example : .1? , .3*($4) , .5+($6)

- the node designated by .1 may be absent;
- the tree designated by .3($4) may be absent or iterated;
- the tree designated by .5($6) must be present and may be iterated;

2.2. Correspondance and constraints

a) Implicit correspondance between tree and forest

To avoid the duplication of the same constraints in the tree part and in the forest part, we allow the following notation facility.

The same node designators in the tree pattern and the forest pattern represent distinct objects related to each other in the following manner :

if C(T.x) is the set of constraints on a node T.x of the tree part and C(F.x), the set of constraints on the node F.x of the forest part, then node T.x verify C(T.x) and the constraints of C(F.x) which are not contradictory with those of C(T.x) (and conversely for node F.x).

This relation may also be explicitly stated for nodes having different designators using the predefined CORRES function.

Some formal constraints linking the tree pattern and the forest pattern are verified at compile time to ensure decidability.

b) Constraints

The constraints part is a boolean expression on labels and decorations of chart pattern nodes. All classical boolean operators are available (and, or, exclusive or, not, imply, equivalent).

Designators are prefixed by A for the tree part and F for the forest part. An expression using non-prefixed designators is interpreted as an expression on the designators of the tree part and of the forest part. The designators of context patterns must be different from the tree part and forest part designators.

Example :

```
CONSTRAINTS       CORRES(T.1, F.4)

    & degre(T.0)=degre(F.4)

    & (    (degre(T.3)^=degre0 & degre(F.1)=degre(F.3))
        V (degre(T.3)=degre0 & degre(.1)=degre(.4))    )
```

2.3. Contexts

A partial mapping described by a chart in a context-free manner may be subordinated to contextual constraints on the left or right context of the described set of sub-strings. This is a powerful tool to describe contextual constraints, co-references, wh-movements, etc. A context element is a sub-string which is described with a corresponding tree pattern.

A tree pattern of the context pattern is a member of a cut of the derivation tree of the context-free skeleton : a context pattern describes a set of cuts in the derivation tree (sect.IV.2).

A context pattern is a forest pattern where each tree pattern may be prefixed by the "not" boolean operator ("^"), indicating the mandatory absence of the tree pattern. Context designators must not be used in other parts of the chart.

Examples : we give some examples of right contexts and their interpretations. The constraint C(.5) is written for a boolean expression on the label and decoration of .5.

- there exists a cut such that the first element of this cut verify C :

  RCTX .5($6) CONSTRAINTS C(.5)

- the first element of each cut verifies C(.5) :

  RCTX ^.5($6) CONSTRAINTS ^C(.5*)

- there exists a cut and there exists an element of this cut such that C(.5) :

  RCTX à4* , .5($6) CONSTRAINTS C(.5)

```
CHART  bx39

TYPE : simple NOUN PHRASES.
CASES : absorption of left and right adverbs.
EXAMPLES : "some of the books",
           "at least two tables".


TREE    .0 ( .1?, $5, .2, $6, .3? )

FOREST  .1? , .0( $5, .2, $6) , .3?

RCTX    .4($7)


CONSTRAINTS

-- constraints on tree nodes :
     etiq(T.0)="nps" & k(T.0)=np
   & sf(T.1?)=des
   & sf(T.3?)=des

-- constraints on forest nodes :
   & etiq(F.0)="np" & k(F.0)=np
   & etiq(F.1?)^="ago"
   & (cat(F.1)=2 a(npmd) V cat(F.1)=2 a(npqtf))
   & sf(F.2)=gov
   & cat(F.3?)=2 a(npmd)
   & ( EXIST(F.1?) V EXIST(F.3?) )
     -- EXIST is a predefined boolean function
     -- testing the existence of an instance :
     -- there must be an instance of .1 or .3

-- constraints on right context and forest nodes :
   & ( EXIST(.3?) =>
       (etiq(.4)^="np" V etiq(F.3)="ago"))
```

Figure 1 : Example of a simple chart for English

Instance of tree and forest patterns for "some of the books":

```
.0  <- "npc".1              .3? <- ∅
.1? <- "some".2             $5  <- "of".3 , "the".4
.2  <- "books".5            $6  <- ∅
```

```
+-----------------------------------------------------------+
!                                                           !
!    bx39                                                   !
!                                                           !
!    TREE                    "nps".1                        !
!                              !                            !
!                 _____________!_____________               !
!                !        !        !        !              !
!                !        !        !        !              !
!          "some".2   "of".3   "the".4   "books".5         !
!                                                           !
!    "npc".1    : deco( k(np), cat(n) )                     !
!    "some".2   : deco( cat(a(npqtf)), sf(des) )            !
!    "of".3     : deco( cat(s(prep))), sf(reg) )            !
!    "the".4    : deco( cat(d), sf(des) )                   !
!    "books".5  : deco( cat(n(nc)), sf(gov) )               !
!                                                           !
!                                                           !
!                                                           !
!    FOREST                        "np".1                   !
!                                    !                      !
!                              ______!_____                 !
!                             !      !     !                !
!                             !      !     !                !
!        "some".2    ,    "of".3   "the".4   "books".5      !
!                                                           !
!    "np".1     : deco( k(np), cat(n) )                     !
!    "some".2   : deco( cat(a(npqtf)) )                     !
!    "of".3     : deco( cat(s(prep))), sf(reg) )            !
!    "the".4    : deco( cat(d), sf(des) )                   !
!    "books".5  : deco( cat(n(nc)), sf(gov) )               !
!                                                           !
+-----------------------------------------------------------+
```

Figure 2 : Chart instance on "some of the books"

IV. THE DERIVATION MECHANISM

1. ELEMENT OF THE MAPPING

An element of the mapping defined by a SCSG is a couple (string, tree) where the correspondance is defined for each sub-tree,

- The string is displayed as a linear graph labeled with string elements (terminals of the grammar).
- The tree is a correspondance tree : to each node is associated a list of paths of the string graph (the correspondance is generally not projective, e.g. representing the "respectivly" construct).

Example of the couple for the string "some of the books" :

```
+-----------------------------------------------------------+
!                                                           !
! bx39                                                      !
!                                                           !
! TREE :                                                    !
!                                                           !
!                        "nps".1                            !
!                          !                                !
!    *       _____________!_____________                   !
!           !        !        !        !                   !
!           !        !        !        !                   !
!       "some".2  "of".3  "the".4  "books".5               !
!                                                           !
! "npc".1    : deco( k(np), cat(n) )          ; (0,4).      !
! "some".2   : deco( cat(a(npqtf)), sf(des) ) ; (0,1).      !
! "of".3     : deco( cat(s(prep))), sf(reg) ) ; (1,2).      !
! "the".4    : deco( cat(d), sf(des) )        ; (2,3).      !
! "books".5  : deco( cat(n(nc)), sf(gov) )    ; (3,4).      !
!                                                           !
!                                                           !
!  STRING :                                                 !
!                                                           !
!  0- "some".2 -1- "of".3 -2- "the".4 -3- "books".5 -4      !
!                                                           !
! "some".2   : deco( cat(a(npqtf)) ) ; (0,1).               !
! "of".3     : deco( cat(s(prep))) ) ; (1,2).               !
! "the".4    : deco( cat(d)        ) ; (2,3).               !
! "books".5  : deco( cat(n(nc))    ) ; (3,4).               !
!                                                           !
+-----------------------------------------------------------+
```

Figure 3 : Aplication of bx39 on "some of the books"

2. DERIVATION IN THE CONTEXT-FREE FRAMEWORK

In the context-free framework, a chart may be seen as a rule in the PROLOG II flavour :

```
+-----------------------------------------------------------+
!                                                           !
!       CHART ( tree, string )       ->                     !
!            TERMINAL(node1, sstring1)                      !
!            <*chart, tree2, sstring2>                      !
!                 .                                         !
!                 .                                         !
!             EVAL(constraints)                             !
!             ARC(tree, string, context) ;                  !
!  ,                                                        !
+-----------------------------------------------------------+
```

- CHART is the chart identifier,
- (tree, string) is the computed couple,
- TERMINAL is a string element definition,
- *chart a variable that will be instantiated with a chart identifier,
- EVAL is a predicate that evaluate the constraints part,
- ARC make the reduction and memorize the contexts for future evaluation.

The algorithm of the context-free skeleton is a bottom-up version of Earley's algorithm defined and used by Quinton <Quinton80> in the KEAL speech recognition system.

For the sake of clarity, the input tape and the factorized stack may be represented as a C-graph. Executing an analysis, the interpreter receives a linear labeled C-graph and works by adding on arcs for each reduced constituent. An arc is labeled by a correspondance tree, the contexts to be evaluated and pointers to the reduced constituents.

Example of a derivation tree for the string "some of the books". The couple calculated is written beside the chart identifiers.

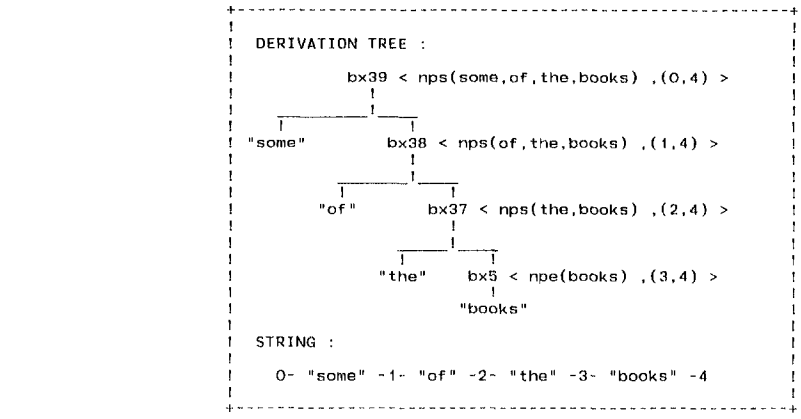


Figure 4 : Derivation for "some of the books"

3. CONTEXT EVALUATION

A context is evaluated during a traversal of the cuts of the derivation tree which go through the node containing the couple (string, tree).

A context pattern memorized on an arc (x, y) describes cuts in the derivation tree. For a C-graph with input node I and output node O, these cuts are represented by paths from y to O (for a right context). A right context evaluation is thus reduced to the search of a sub-C-graph from y to O.

V. CONCLUSION

In the study and developement of future industrial MT systems, the definition and implementation of linguistic parts raise many problems partly due to the size of the systems, and we hardly need rigorous methods of developement. The challenge is to theorize the line of activities which go from the formal specification of an informal problem to the proof of correctness of the corresponding program, and this in the field of linguistic programming.

As a first step in this direction, within the specification/implementation/validation framework, a software environement is being developed based on a linguistic specification language.

This language allows linguists to specify analysis and generation modules for natural language sub-sets and the interpreter allows empirical validation of implementations. The language is actually used to specify the grammars of the French MT National Project, and for all new developements carried out in GETA.

For further work, extensions are studied, specially the specification of transfer modules. Work is also being engaged to develop rigorous linguistic programming methods using SLLP provided with SCSG specifications.

REFERENCES

<Boitet85> : BOITET C., GUILLAUME P., QUEZEL-AMBRUNAZ M. A case study in software evolution : from ARIANE 78.4 to ARIANE 85 Proceedings of the Conference on Theoretical and Methodological issues in Machine Translation of Natural Languages, Colgate University, Hamilton, N.-Y., U.S.A, August 14-16, 1985.

<Kay84> : KAY M. Functionnal Unification Grammar : a formalism for Machine Translation Proceedings of COLING-84

<Maas84> : MAAS H.-D. The MT system SUSY presented at the ISSCO Tutorial on Machine Translation, Lugano, Switzerland, April 2-6, 1984.

<Nakamura84> : NAKAMURA J., TSUJI J., NAGAO M. Grammar Writing System (GRADE) of Mu-Machine Translation Project and its caracteristics Proceedings of COLING-84.

<Quinton80> : QUINTON P. Contribution à la reconnaissance de la parole, Utilisation de méthodes heuristiques pour la reconnaissance de phrases Thèse de Docteur ès Sciences en Mathématiques, Université de Rennes, 1980.

<Slocum84> : SLOCUM J. METAL : the LRC Machine Translation System presented at the ISSCO Tutorial on Machine Translation, Lugano, Switzerland, April 2-6, 1984.

<Vauquois85a> : VAUQUOIS B., BOITET C. Automated Translation at Grenoble University Computational Linguistics, 11/1, pp 28-36, January-March 1985.

<Vauquois85b> : VAUQUOIS B., CHAPPUY S. Static Grammars: a formalism for the description of linguistic models Proceedings of the Conference on Theoretical and Methodological issues in Machine Translation of Natural Languages, Colgate University, Hamilton, N.-Y., U.S.A, August 14-16, 1985.

<Yan86> : YAN Y.-F. Structural Correspondance Specification Environement Proceedings of COLING-86

<Zaharin86> : ZAHARIN Y. Strategies and Heuristics in the Analysis of a Natural Language in Machine Translation Ph.D. Thesis, Universiti Sains Malaysia, Penang, March 1986. Research work under the GETA-USMG Cooperation (GETA document).

-o-o-o-o-o-o-o-o-