ATN COMPILER AND PARSER FOR AN ASR SYSTEM

Emmanuel REYNIER and Jean CAELEN

"Institut de la Communication Parlé" UA n° 368
INPG / ENSERG - Université Stendhal 46, Av. Félix Viallet F-38031 Grenoble Cedex

ABSTRACT

This paper describes some aspects of the linguistic expert system in continuous speech recognition, which makes part of the DIRA (Integrated Dialogue and Automatic Recognition) project, being developed in our laboratory. This system consists in a rule compiler and an analyzer.

The compiler accepts context free or context sensitive rules and transform rules. It produces a transition network (ATN) which contains the nodes corresponding to the syntactic, lexical or phonetic categories and the transition arcs, whose traversal is conditioned by rules and predicates. These conditions arise both from semantic attributes for syntactic semantic analysis, and from phonologic rules describing the phonetic network.

In fact, it is possible to produce one multi-layer network in which interactions between syntactic-semantic level and lexical level are strong. For example, one chooses the input level of the grammar (axiom) and the description level of the terminal vocabulary. One can define also phonologic rules constrained by syntax or syntactic constraints from lexical informations, etc ... 

INTRODUCTION

Speech is a temporal signal that looks like a discrete data sequence after segmentation into acoustic units. This sequence is arranged at least formally in an ordered symbol series respecting a kind of syntactic constitution. Accordingly, speech offers a double structure: (a) hierarchical -- on a paradigmatic axis -- from acoustic units to sentences crossing over phonemes, syllables, words, syntagmes, etc. and (b) sequential -- on a syntagmatic axis -- among each hierarchical level: there is a sentence syntax with word units described in grammar rules, such as a word syntax with phonemes described in phonologic rules. Speech recognition systems widely use this property [2], [7] to standardize the network idea throughout the analysis levels, from APD to semantic. Likewise the DIRA system [11] joins with this idea.

In these cases, an ATN compiler can be used to produce a network by means of formal description of these units and corresponding grammars. ATN or equivalents are known as among the most powerful tools used to resolve problems of syntactic-semantic representation and analysis. The DIRA system already uses ATN in the APD [8]. For the purpose of the generality and coherence of the recognition system, it is a prime necessity to make these approaches "network" uniform.

This paper describes the input units, the grammar, the compiler itself, and the compiled network. Examples are taken from phonetic, phonologic, lexical, syntactic or semantic domains in order to maintain a general approach and to keep in mind the need to integrate the specificity of each domain.

1. DESCRIPTION OF THE COMPILER INPUT UNITS

The compiler receives as input the grammar G and the lexicon L of the application. It accepts the most general type of grammar in order to provide for the widest possible application. It can therefore be used to manipulate both context sensitive, transformation and functional lexical grammars. A grammar G is the 5-tuplet:

\[ G=(A, \text{VN}, \text{VT}, \text{VA}, \text{R}) \]

- A: axioms' set,
- VN: non terminal vocabulary,
- VT: terminal vocabulary
- VA: auxiliary vocabulary
- R: rules set describing the grammar G.

(by convention \( V = \text{VN} U \text{VT} U \text{VA} \), \( V^+ = \text{free monoid generated by} \text{VN} \), \( V^* = \text{free monoid generated by} \text{VT} \) )

These rules are matched with two additional fields: a "context" field and an "action" field. Their general form is described in the description language \( \mu(G) \). Likewise the lexicon is written in the description language \( \mu(L) \).

Other notation: 

Let \( <> \) be a character string and \( \langle \rangle \) an integer, the following variables indicate:

- \( G<<e> \): grammatical element in left part of rule
- \( E<<e> \): rule labeled \( <> \)
- \( R<<e> \): register named \( <> \)
- \( Cs<<e> \): syntactic category named \( <> \)
- \( S<<e> \): semantic category named \( <> \)
- \( As<<e> \): attribute named \( <> \)
- \( Ns<<e> \): lexical instance named \( <> \)
- \( Ps<<e> \): lexical instance similar to the prototype \( <> \)
- \( Ms<<e> \): morpho-phonologic rule named \( <> \)

1.1. Grammar description language \( \mu(G) \)

This language allows the description of the application grammar in a formalism commonly used by linguists and using a notation similar to the Bachus-Naur one. It is separated into:

- the vocabularies declaration part \( \mu(Dv(G)) \)
- the rules part \( \mu(R(G)) \)

1.1.1. The hierarchy of vocabularies

The compiler must be advised about the element of vocabularies corresponding to the different levels of description used in grammar \( G \). These classes have to be declared in a hierarchical structure in order to take into account the order relation between constituents like acoustic units, phonemes, words, etc. This hierarchy, called \( \mu(Dv(G)) \), is a parenthesized structure as:

\[ \{ Nc1; C1 ( Nc2; C2 ( Nc3; C3 ... ( Ncn; Cn ))) \} \]

\( Nc1 \): class name at the overlapped level 1
\( C1 \): list of class \( Nc1 \)'s constituents

Outside this structure, the two classes \( T(\text{Terminal}) \) and \( NT(\text{non terminal}) \) always exist. They contain the elements out of all classes which never appear in the left part of non transform rules and the complementary class respectively.

Before each analysis it is necessary to fix the high level class which corresponds to the set of axioms and the low level class which corresponds to the terminal vocabulary of the current analysis. The analyzer chooses dynamically the analysis level with respect to the values of the two following command variables:

- the variable \$AXIOM: points to the class whose elements make (part of) the axiom set A or by default the left element of the first non transform rule.

---

EUROSPeech '89, Paris, France, September 1989

1398
are instantiated from hierarchically organized classes which inherit lexical description: each input The lexicon's instances are related to these two structures: it permits their ascendant's properties. The classes are double hierarchical organized: (a) within a syntactic relation and (b) a semantic relation.

In the ATN, there is no longer any need to be concerned with the concept of context. They are activated on crossing network whose part is to pick out the elements' position in a sentence. The left (resp. dominant, right) context is conventionally precede by the sign '<' (resp. '>', '>').

### 1.1.2. The variables (registers)

These allow the manipulation of all the objects in the grammar and lexicon inside the ATN [10]. Their type is free of declaration, and is fixed by the use you make of it. This notion is very general and extends to concepts of value, pointer, predicate and procedure. It is not necessary to declare them because they are all marked by the prefix R3 (see example 8).

### 1.1.3. The rules

Each rule is written in respect of the following syntax µ(G):

E: X op Y < a ^ b > c / Ac;

where:

- E is the optional rule's identifier.
- X is the left part of the rule such as X e Vn, or X e V* for transformation rules.
- op=[-, >, >>, >>>] with the following conventions:
  - -> for non transform rules
  - => for optional transform rules
  - >> for obligatory transform rules

The rules formally described above have two particular fields, "context" and "action", whose part (role) is explained below.

### 2. THE WORKING OF GRAMMAR RULES AND THEIR RELATIONS TO THE LEXICON

The rules formally described above have two particular fields, "context" and "action", whose part (role) is explained below.

#### 2.1. The contexts

A context is an element of Ψ including some grammatical marks, binary operators ' (' (strict coordination) and ') ' (broad coordination) and unary operator ' - ' (negation). These binary operators represent respectively strict or broad sequenced relation. The grammatical marks are HEAD(X) and TAIL(X) with X e Vn, whose part is to pick out the elements' position in a sentence. The left (resp. dominant, right) context is conventionally precede by the sign '<' (resp. '>', ' ').

Example 3: problem of the location of the personal pronoun in French inside the enonciative sentence. 

P -> [SP] SN SV [SN];
SN -> Pro_pers ^ P > SV;

"Dans la forêt il mange un lapin." is a correct sentence, with regard to the location of the personal pronoun.

#### 2.2. The actions

The concept of action is more widely used with ATN than the concept of context. They are activated on crossing network transitions. These are premises, that is to say well formed logical expressions (ELBF), which take the values "true" or "false". They are specified in the rules and evaluated at the time the analyzer arrives at the corresponding transitions which are validated if they are true. The terms of the premises are predicates P or evaluable
predicates Pe that is to say procedures which return the value L(Pe)="true" or "false" after execution. For the purpose of compact representation it is possible to write actions -- as in the C language -- in the following manner:

\[ ELBF(P, Pe)=(P1+P2.Pe3)+P4 \]

which is equivalent to the following algorithm:

Begin execution Pe
  if L(Pe)="true" then validate transition
  else if P2.Pe="true" then execution Pey
  if L(Pe)="true" then validate transition
  else if P2.Pe="true" then validate transition
  else inhibit transition
End

This way of expressing the tests and the procedures enables an interesting coding of the actions writing attached to transitions. Each transition can be governed by an option actionally empty that is always true. The rule's action list Ac can therefore be written:

\[ Ac = ELBF1, ELBF2, ..., ELBFn \]

for a rule:

\[ X \rightarrow Y_1 Y_2 Y_3 ... Y_n \]

Into the ATN we get the implantation:

```
  X
 /\  Y_1
 /  \ ELBF
  \  Y_2
  \ /
  Y_n
     ELBF
      X
```

Among the defined procedure and predicates known by the compiler, you can hold:

- predicates:
  - @: always 'true' (empty action)
  - TEQ(Xa,Xb): 'true' if the variable Xa is equal to Xb, etc...
- arithmetic procedures:
  - ADD(Xa,Xb,Xc): add of Xb and Xc and store the result in Xa
  - ING(Xa) increment Xa
  - ASG(Xa) assign Xb to Xa, etc...

These procedures return 'true' after execution.

The syntactic-semantic analysis is always connected to the lexicon in accordance with:
- the verification -- where it is a matter of providing a list of possible candidate words
- the prediction (very important for speech processing)
- the matching procedures:

```
MATH(Xa,Xb,Xc): true if the attributes Xa and Xb's value is compatible in the variables Xa and Xb
UNIF(Xa,Xb,Xc): unify the variables Xa and Xb's attributes (Xa prevailing the Xb) and return the unified Xb's attributes.
```

Example 4:

GN -> N Adj @@, TEQ(D4(A$gender),D2($A$gender))

This rule is completed if the noun and adjective's gender are compatible. You then return the result of unification between N (D4) and Adj (D2) to GN (G5).

2.3. Lexical access constraints

The syntactic-semantic analysis is always connected to the lexicon in accordance with:
- the verification --where it is a matter of accepting or rejecting a word sequence-- needs a lexical access to look at the syntactic-semantic attributes of the specified words.
- the prediction (very important for speech processing) --where it is a matter of providing a list of possible candidate words after a correct sequence-- needs a lexical access too from syntactic-semantic attributes predicted by the analyzer but examining all the possible paths into the ATN.

In these two cases it is evident that the relation between syntax and lexicon is very strong and it can be anticipated at the compilation time to reduce the access time (precompiled static access). This relation is taken into account by constraint access which gives information on syntactic or semantic categories, on attributes values, etc. Obviously some actions could resolve the same problem but the execution time would be longer because of the access calculated in every instance. These constraints are pointed out directly in the rules after each term if necessary.

Constraints of the access on terminal vocabulary V1

There are 4 types of lexical constraints that you associate directly on elements of the terminal vocabulary:
- on syntactic category: for example Det(C$def) force the determiner to be of the definite category,
- on semantic category,
- on attributes' values,
- on lexical instances.

These constraints can limit the lexical search in prediction mode and restrain a rule's application in verification mode.

Example 5: Access to fixed subclasses or lexical instances

\[ GP \rightarrow PREP(A$de, A$non-propre, S$ville) \]

constrain the rule to match only the prepositional groups whose preposition is "de" or "a" and noun is a name of town. During compilation a direct link is created between the right term PREP and the lexical instances "de" and "a" as the term N and the syntactic and semantic categories "non-propre" and "ville".

Constraints of the access on non terminal vocabulary V2

This case is more general because it deals with meta rules and therefore makes it possible to point out constraints on rules themselves after each non terminal term.

Example 6:

1. GV -> V GP(D51(1$en))
2. GP -> PREP A$de, A$non-propre, S$ville

This theoretical example shows that rule 1 is applicable only if GP's first term (PREP) is "en".

Example 7:

This example shows the different possibilities offered by the grammar's description language:

RULES:

| a | P -> P_positive;
| b | P -> P_negative;
| c | P -> P_imperative;
| d | P -> P_formal (P$en);

\[ ASG(\text{PREP}, \text{indicatif}, 1, 1, 1, 1, 1, 1, 1) \]

Example 8:

This example shows the different possibilities offered by the grammar's description language:

RULES:

| a | P -> P_positive;
| b | P -> P_negative;
| c | P -> P_imperative;
| d | P -> P_formal (P$en);

\[ ASG(\text{PREP}, \text{indicatif}, 1, 1, 1, 1, 1, 1, 1) \]

Example 9:

This example shows the different possibilities offered by the grammar's description language:

RULES:

| a | P -> P_positive;
| b | P -> P_negative;
| c | P -> P_imperative;
| d | P -> P_formal (P$en);

\[ ASG(\text{PREP}, \text{indicatif}, 1, 1, 1, 1, 1, 1, 1) \]
3. NETWORK’S IMPLANTATION

The grammatical network is a graph whose nodes represent the elements from the set V and whose arcs represent the transitions which holds the actions defined into the rules. Each non terminal element x ∈ Vn appears as a pair of nodes: HEAD(x) input node and TAIL(x) output node. Each terminal element appears as only one node. After compiling, only one network is produced and so there are no calls to a sub network: thanks to this implantation, the context and transform rules are workable because each non terminal element appears as only one execution from left to right so an analysis from right to left is chained link between nodes. However actions are written for an execution from left to right so an analysis from right to left starting at a anchoring place is realized in two states: (a) looking for one or more nodes on the left and then (b) running the analyzer from left to right (Fig. 2).

In a general view the implementation of a non transform rule can be represented in a form like figure 4 where Y2 belong to Vn:

![Figure 2: basic cell representing a node and its associated transitions.](image)

<table>
<thead>
<tr>
<th>Head</th>
<th>Y2 ∈ Vn</th>
<th>Y1 ∈ V*</th>
<th>change of direction</th>
<th>A: bottom-up mode</th>
<th>D: top-down mode</th>
</tr>
</thead>
</table>

When an element appears more than once in the grammar, it is not duplicated in the ATN. So a node is into an environment reflecting all the contexts where the corresponding element is. The analyzer therefore has an efficient ascendent operation.

4. PATH THROUGH THE ATN DURING ANALYSIS

An analysis consists in going through the ATN depending on the working fixed by the supervisor [1]. This is done by the network controller, called analyzer for simplicity. Two modes of analysis are proposed: the bottom-up mode and the top-down mode. At any time two workings are possible for each mode: (a) a verification working and (b) a prediction working. To verify an input form the analyzer look for a path into the ATN starting from the current node. In prediction working, the analyzer propose all the possible successor nodes to the current one at a distance k.

Beginning at syntactic land (anchoring) marks as begin or end sentences, the top-down analysis is well suited. On the other hand when the analyzer does not know the current syntactic position but lexical land marks, then the bottom-up analysis is activated. The analyzer authorizes the left-right or right-left path direction. So it can perform an analysis from middle to sides starting on land marks.

The analyzer constructs all the syntactically and semantically correct solutions in parallel, from lexical information and used actions. When the analysis is completed you have a tree of syntactically and semantically correct solutions (constituents’ structure) and functional solutions [5]. The number of rules does not prejudice the speed of analysis but only the branching factor on each node. The analysis of the sentence “Le petit lapin de garenne ne l’a pas vu.” from the grammat of the example 7 gives:

![Figure 3: general implantation of a non transform rule](image)

When an element appears more than once in the grammar, it is not duplicated in the ATN. So a node is into an environment reflecting all the contexts where the corresponding element is. The analyzer therefore has an efficient ascendent operation.

5. CONCLUSION

The compiler accepts several kinds of grammar: context free, context sensitive, transformational and functional lexical. this enables syntactic specialist to write flexible grammars oriented to specific applications: syntax of restricted languages, phonology of French, morphology, etc. without restriction to a particular linguistic society, pp. 142-158: 1979