ATN COMPILER AND PARSER FOR AN ASR SYSTEM

Emmanuel REYNIER and Jean CAELEN

"Institut de la Communication Partielle" UA n° 368
INPG / ENSERG - Université Stendhal 46, Av. Félix Viallet F-38051 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 these properties [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:

\[
G = (A, V_n, V_t, V_a, R)
\]

where:
- \( A \) : axioms' set,
- \( V_n \) : non terminal vocabulary,
- \( V_t \) : terminal vocabulary,
- \( V_a \) : auxiliary vocabulary.

\( R \) : rules set describing the grammar \( G \).

(by convention \( V = V_n U V_t U V_a \), \( V^* = \) free monoid generated by the vocabulary \( V, V^a+V^* \))

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 \( \langle \rangle \) be a character string and \( \langle \rangle \) an integer, the following variables indicate:
- \( G_{\langle n \rangle} \) : grammatical element in left part of rule
- \( E_{\langle n \rangle} \) : grammatical element in right part of rule
- \( ES_{\langle n \rangle} \) : rule labeled \( \langle \rangle \)
- \( R_{\langle n \rangle} \) : register named \( \langle \rangle \)
- \( CS_{\langle n \rangle} \) : syntactic category named \( \langle \rangle \)
- \( S_{\langle n \rangle} \) : semantic category named \( \langle \rangle \)
- \( A_{\langle n \rangle} \) : attribute named \( \langle \rangle \)
- \( I_{\langle n \rangle} \) : lexical instance named \( \langle \rangle \)
- \( PS_{\langle n \rangle} \) : lexical instance similar to the prototype \( \langle \rangle \)
- \( M_{\langle n \rangle} \) : morpho-phonologic rule named \( \langle \rangle \)

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 \( \mu(D\mu(G)) \)
- the rules set 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 \( \mu(D\mu(G)) \), is a parenthesized structure as:

\[
\{Nc_1: C_1 \ Nc_2: C_2 \ Nc_3: C_3 \ldots \ Nc_n: C_n \ldots \}
\]

\( Nc_i \) : class name at the overlapped level \( i \)

\( C_i \) : list of class \( Nc_i \) 's constituents

Outside this structure, the two classes \( T(Terminal) \) and \( NT(non \ terminal) \) 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 \( S_{AXIOM} \) : 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.

Fig. 1: the ATN compiler
- the variable STERM_VOC: points to the class whose elements make the vocabulary Vt in conjunction with the class T. VA contains all the symbols that belong to the classes whose level is lower than STERM_VOC one. Then $V_n = C(Vt \cup VA)$ with C being the complement operator.

Example 1: context free grammar definition
(PHRASE: P-infinite, ...
(SYNTAXME: SN, SP, ...
(CAT_SYNTAX: Verbe, Nom, Dts, Prtp, Adj, ...
(PHONEME: @, p, t, k, f, s, i, o, u, l, ...

1.1.2. The variables (registers)

These allow the manipulation of all the objects in the grammar and lexicon inside the ATN [110]. 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 R$<$ (see example 8).

1.1.3. The rules

Each rule is written in respect of the following syntax µ(R): E: X op Y < a ^ b c > Ac;
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 transform rules,
op=[->, =>, =>>] with the following conventions:
- for non transform rules
  => for optional transform rules
  =>> for obligatory transform rules
Y is the right member such as Y e V*,
a is the left context optionally empty, such as $a e V^*$,
b is the dominant context optionally empty, such as $b e V^*$,
c is the right context optionally empty, such as $c e V^*$,
Ac is an optionally empty action list which contains as many fields
separated by commas as there are in the right member plus one,
(1: -> => >> < ^ > ...) are separators attached to rules fields.

Notes:
- 1- the notation X X -> Y < a > b is equivalent to the more classical notation $a X b$;
- 2- right member can be in the Bachus-Naur Form (BNF).
- 3- all recursive forms are accepted.
- 4- you can join a label with a rule to refer to it more easily.

1.2. Description Language of the lexicon µ(L)

This language follows a frame model, i.e. the lexical elements are instantiated from hierarchically organized classes which inherit their ascendant's properties. The classes are double hierarchical organized: (a) within a syntactic relation and (b) a semantic relation. The lexicon's instances are related to these two structures: it permits the separation of the syntactic from the semantic inputs in order to have clearer and more efficient access method. This model approaches the formalism of Functional Description [6].

After the compiler has used the lexicon to set up pointers into the ATN, there is no longer any need to be concerned with the specific contents of the lexicon. We can come back later to this mechanism. This example shows the main characteristics of the lexical description: each input is associated with an orthographic and a phonetic representation, attributes fit with facets which define their type and their domain. Some objects can be declared to have the same construction as mentioned in LFG grammar [5]. The procedural part (for instance morpho-phonological rules) enables to represent derived forms of the lexical units.

Example 2: instantiated tree of the word "lapin"

LEXICON:
[ Lapin1: orth="lapin"
  phon="lapez-
  est_un SSrongeur
  habitat=[sauvage, domestique]
  complement=[Sespece, SCouleur]
  est_un CSNom
  genre=masculun /M$R2]
  nombre=singulier /M$R2]
  [ Lapin2: est comme ISLapin1
  est_un $SVaine
  preparation=[gril, four, sauce]
  ...
  "rongeur: some de SSmannifere
  taille=Smesure(AIntervalle(5, 50),ASunite(cm))
  poids=Smesure(AIntervalle(0, 2, 5),ASunite(kg))
  [ SN: some de CS
  dominant=[CNome, CSPronom]
  quantifieur=CNomigaQuantifieur]
  genre=[CNome, CNom]
  nombre=[ANomina, ANombre]]
  [ Nom: sorte_de CS]
  complement=CSSP
  quantifieur=CSSDet
  genre=[masculin, feminin]
  nombre=[singulier, plural]
]

MORPHO-PHONOLOGICAL RULES:
[ 1: */ rule for the feminine's construction */
  *feminin= orth: x > xe;
  phon: e-> > in]

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 V* including some grammatical marks, binary operators '+' (strict coordination) and '-' (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 preceded 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 foret 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 promises, 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 promise 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 :

\[
\text{ELBF}(P, Pe) := (P1 + P2, Pe3) + P4
\]

which is equivalent to the following algorithm:

Begin

 execution Pe1
 if L(Pe1)="true" then validate transition
 else if P2="true" then execution Pe2
 if L(Pe2)="true" then validate transition
 else if P3="true" then validate transition
 else inhibit transition
 endif

 execution Pe3
 if L(Pe3)="true" then validate transition
 else if P4="true" then validate transition
 else inhibit transition
 endif

End

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

\[
Ac = \text{ELBF}(P1, \text{ELBF}(P2, \text{ELBF}(P3, \text{ELBF}(P4, \text{ELBF}(P5, \ldots, \text{ELBF}(Pn)))))
\]

for a rule ax:

\[
X \rightarrow Y_1 Y_2 Y_3 \ldots Y_n / \text{ELBF}(P1, \text{ELBF}(P2, \text{ELBF}(P3, \text{ELBF}(P4, \text{ELBF}(P5, \ldots, \text{ELBF}(Pn))))).
\]

Into the ATN we get the implantation:

\[
\text{HEAD}(X) \text{ ELBF}_1 \text{ ELBF}_2 \ldots \text{ ELBF}_n \text{ TAIL}(X)
\]

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
    - ASG(Xa, Xb) assign Xb to Xa, etc . . .
  - these procedures return 'true' after execution.
  - the matching procedures:
    - MATCH(AI(Xa, Xb)); 'true' if the attribute Att's value is compatible in the variables Xa and Xb
    - UNIF(As, Xa, Xb); unify the variables Xa and Xb's attributes (Xa prevailing Xb) and return the unified structure in Xa.

Example 4:

\[
\text{GN} \rightarrow \text{N Adj} @ @ \text{TEQ}(\text{ADJ}(\text{A$gender}), \text{DS2}(\text{A$gender})).
\]

This rule is completed if the noun and adjective's gender are compatible. You then return the result of unification between N (DS1) and Adj (DS2) to GN (GS1).

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 semantic 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

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

\[
\text{GP} \rightarrow \text{PREP}(O$,de, a) \text{ N}(\text{non-propre, S$ville});
\]

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

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: \text{GV} \rightarrow \text{V GP}(\text{DS1}(\text{I$en}))

2: \text{GP} \rightarrow \text{PREP} \text{ GN}

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:

- P \rightarrow P_{affimative};
- P \rightarrow P_{imperative};
- P \rightarrow P_{Coj}([\text{Coordination}] P);
- P \rightarrow \text{SV}([\text{fixed subclasses or lexical instances}]);
- P \rightarrow \text{SN} \rightarrow \text{Pronom}(\text{C$personne})$

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

Example 5: Access to fixed subclasses or lexical instances

\[
\text{GP} \rightarrow \text{PREP}(O$, de, a) \text{ N}(\text{non-propre, S$ville});
\]

- on lexical instances.
- on syntactic category: for example Det(C$def) force the determiner to be of the definite category,
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 implementation, the context transform rules are workable because all the possible paths through the network are explicit.

The ATN is constructed in a way such as to enable the analyzer to go left to right as well as right to left: so there is a double linked chain between nodes. However actions are written for an execution from left to right so an analysis from right to left starting at an 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).

![Diagram](image)

In general view the implementation of a non transform rule can be represented in a form like figure 4 where Y1 belong to Vn:

\[ X \rightarrow Y / \alpha_0, \alpha_1, \ldots, \alpha_n; \text{ with } X \in Vn \text{ and } Y \in V^* \]

When an element appears more than once in the grammar, it is not duplicated in the ATN. So a node is in an environment reflecting all the contexts where the corresponding element is. The analyzer therefore has an efficient ascending 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 grammar of the example 7 gives:

Analysis in constituents:

\[ PF \rightarrow \text{affirmative} SN( \text{Det(}I\text{se}) SN(2) S Adj( Special) SN(2) Nnom(1) (Ispanin1)) \]  
\[ SP( \text{Prep(}I\text{de} SN(2) (Nnom(Ispanin)))(S) S V( Aux(}I\text{se} GV(2} \text{ Verbe}(}I\text{vo}) SN(1) B31)) \]

Forme=Negative, Mode=Indicatif

Functional analysis:

(ieff=Negative, Mode=Indicatif)

(Def=

Forme=Negative, Mode=Indicatif)

Type=defini)

Genre=mascarin)

Nom(1)

quantifier=(Ise)

Module=(Lse)

zelf=petit)

Sujet=(Ispanin)

complement=(Ispanin)

type=defini)

genre=mascarin)

nomm=vectorize)

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, morphotaxy, etc. without restriction to a particular formalism. This compiler is written in C language and the structural objects in C++. The compiling time increases linearly with the number of rules so that the compiler is a tool usable in real applications like speech recognition.

6. REFERENCES


EUROspeech '89, Paris, France, September 1989