COLING 82, J. Horecký (ed.)
North-Holland Publishing Company

ARBUS, A TOOL FOR DEVELOPING APPLICATION GRAMMARS

D. MEMMI J. MARIANI

LIMSI (CNRS) B.P.30
91406 ORSAY FRANCE

The development of a natural language system usually requires frequent changes to the grammar used. It is then very useful to be able to define and modify the grammar rules easily, without having to tamper with the parsing program. The ARBUS system was designed to help develop grammars for natural language processing. With this system one can build, display, test, modify and file a grammar interactively in a very convenient way. This was achieved by packaging a parser and a grammar editor with an elaborate interface which isolates the user from implementation details and guides him as much as possible.

INTRODUCTION

Parsing is one of the main problems in natural language processing. It is generally recognized that understanding written text requires some kind of structural analysis, even if semantic comprehension would also be needed. In speech recognition research, syntactic constraints are frequently used to help acoustic recognition by reducing the number of possibilities to be dealt with.

Grammatical analysis becomes even more important when one considers it to include not only syntax itself, but also any formal constraint. One can therefore define semantic or pragmatic grammars, dialog grammars as well as phonetic or phonological rules. The formalism of syntactic rules is powerful enough to describe many areas of natural language beside syntax itself, and the use of grammar has been extended accordingly in many systems.

But parsing is a difficult problem. The design of a parser involves fairly sophisticated programming techniques. And grammar rules are usually numerous, and their interaction may prove quite complex, so that it is not easy to define a grammar. Rules often have to be modified repeatedly for the development of the grammar, and will have to be modified again if one wants to change the application domain.

To avoid tinkering constantly with the program, rules should be kept separate from the control mechanism of the parser. The grammar is then considered as data for the parsing program, and the rules can be given in a clear declarative formulation. If this basic precaution is not observed, modifying the rules will require repeated and tiresome reprogramming, and at some point the program may become too complicated for any further extension of the grammar.

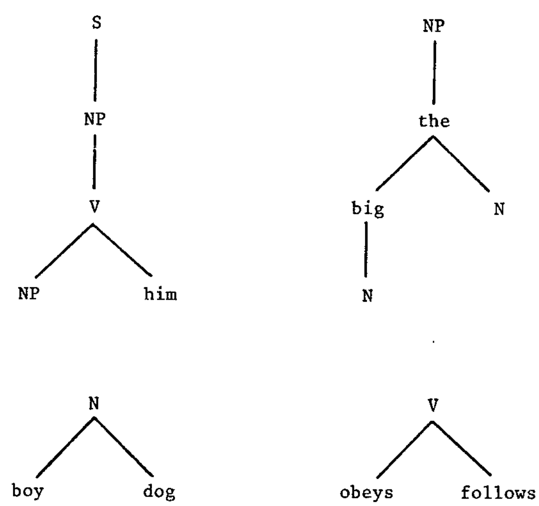
It is also very interesting to help the user to develop his own grammar, by allowing him to define, test and modify the rules easily thanks to a specialized interactive interface. For example the LIFER system (Hendrix, 1977) was specially designed to help build application grammars without extensive programming. It has been successfully used to build complex natural language front-ends such as the

LADDER system (Sacerdoti, 1977; Hendrix and al., 1978) in order to access and query databases in natural English.

Similarly we have designed the ARBUS system as an aid to the development of grammars. With ARBUS one can build, display, test and modify a grammar interactively in a very convenient way. The user never has to deal directly with the underlying programs and no programming is necessary. This was achieved by packaging a parser and a grammar editor with an elaborate interface, which shields the user from implementation details and guides him as much as possible.

GRAMMAR AND PARSER

A grammar is implemented in ARBUS as a set of trees, with a tree for each syntactic category. Each node of a tree (except the root) represents either a terminal word of the language defined, or a category referring then to another tree (fig. 1). This is also the way the user must describe a grammar to the system. This representation is a simplified form of transition networks, where each subnetwork corresponds to a different syntactic category.



S --> NP V NP
S --> NP V him
N --> boy
N --> dog

NP --> the big N
NP --> the N
V --> obeys
V --> follows

Fig. 1. Transition trees and corresponding rules

A tree structure is generally less compact, but absolutely equivalent to a network (by duplicating nodes with multiple parents in the related network). We chose this representation because trees are easier to describe and to visualize interactively. They are also easier to process and to display than unrestricted graphs. And every distinct path in a syntactic tree corresponds to a rewrite rule of the grammar, which is not true in general for transition networks.

Any node can be augmented with tests and actions to be performed when coming across the node. These tests and actions are predefined in a library at the disposal of the user, and each one is known under a reference name so that they can be used without having to deal with their actual implementation. For instance, there is an action available to note that a noun phrase is singular, and a test to check later on that the subject of a verb was indeed singular. Another action translates a sequence of digits into the corresponding number, etc...

These augmentations make it possible to define context-sensitive languages, as one can take the context into account with actions and tests, in order to handle conveniently features such as number agreement between subject and verb. This representation of grammars is then quite similar to Augmented Transition Networks (Woods, 1970), in which tests and actions can be associated with the transitions. The main difference is the use of trees instead of networks to implement a grammar in our system.

The parser which will test a grammar by interpreting its representation is also comparable to an ATN parser. It is designed as a top-down, left-to-right parser: when moving through a tree, control is transferred to another tree every time a syntactic category is encountered at a node. This process can be recursive thanks to a pushdown stack. If at a given point there are several possible paths, the parser follows only one, but saves the current state on the stack and will backtrack in case of failure to try the alternatives.

If a node is augmented with a test, the transition can be followed only if the test is verified; if there is an action at the node, the action is performed (but will be undone in case of backtracking). The actions could be used to build the parse of a sentence, but in fact the parse-tree produced is simply a trace of the successful transitions through the grammar if the sentence is accepted. This is actually closer to the way a context-free parser operates. If a sentence is ambiguous, one version of the parser returns only one analysis; another slower version produces all the possible parses.

If the input sentence is not accepted, the parser tries to give a simple and clear diagnosis of the failure and specifies the place in the sentence where it had to give up. But systematic backtracking sometimes makes it difficult to tell exactly what happened; it might be useful to save the whole parse history. Lastly the parser can also run in predictive mode for speech recognition: the grammar is user to constrain possibilities at every step to help lexical recognition.

The grammar can also be employed to generate sentences. A special generator using a random function produces sentences according to the current grammar. This quickly gives a broad view of the type of language defined, without using the parser and without having to think up successive sentences to test. The random generator offers then one more facility to examine a grammar and sometimes reveals unforeseen errors in the syntactic rules.

So by and large, parsing is done in ARBUS with fairly standard tools which are comparable to other well-known parsers. But the emphasis was put mainly on practical interactive use to develop an application grammar, and most design decisions were taken with this primary goal in mind.

GRAMMAR EDITOR

To define a grammar , the user describes it to ARBUS in the form of transition trees as seen above. Each tree is to be described by moving through the tree in depth-first fashion from left to right, with the help of a prompting program. The system then builds the corresponding internal representation. Actions and tests can also be added on the nodes. But after testing the grammar with the parser, it will often appear necessary to modify the syntax. One must therefore be able to edit the grammar.

We designed a specialized grammar editor containing a complete set of diplay and modification functions. Because of the way the grammar is represented within the system, this editor deals mainly with tree structures. We tried to select a minimal set of primitives that would allow all the necessary modifications while being simple to learn. More complex editing operations may then have to be executed in several steps.

The grammar can first be displayed, as a whole or tree by tree, with actions and tests if needed. One can either display the trees themselves, or list all the distinct paths of a tree, which correspond to rewrite rules. The lexicon may also be examined, as well as the list of syntactic categories of the grammar. The lexicon is automatically updated after any modification and thus always shows the current state. One can also look up the catalogue of actions and tests available to the user for augmentations.

With the editor one can replace one word by another, whether at a given node, in a whole tree or everywhere in the grammar. To modify the structure of a transition tree, one can delete, insert or replace a node by itself without its offspring, or a node with its offspring (i.e., a sub-tree). It is also possible to save part of a tree to insert it elsewhere. If a new syntactic category is introduced during a modification, the system will detect it and ask for the description of a new transition tree.

Augmentations can of course be also modified by adding, deleting or replacing tests and actions at any node. In short everything in the grammar may be examined and modified. When the result seems satisfactory, the grammar can then be saved on file. It may be recalled later for another session of testing and modifications, used for an application, or even be sent to another parsing system.

This editor is fairly simple, and more complex functions could be added. But it allows any possible modification of tree structures and already includes a certain number of functions. How to use the editor is then not immediately obvious, and to help the user all editing functions are in fact packaged within a special interactive interface. Modifications will be performed through this interface, which will be responsible for all interactions with the user.

USER INTERFACE

Because ARBUS is intended primarily to be a development aid, the user interface was designed with particular care and constitutes a sizable part of the whole system. Without this interface, the large number of construction, parsing and editing functions available would have required a detailed instruction manual and a long training period to use the system fully.

The basic principle followed in the design of the interface is then to guide the user as much as possible through an interactive dialog at the terminal. The interface totally isolates the user from underlying programs and redefines its own environment regardless of the implementation language. All system functions will be called only by typing commands to the interface, which acts as a command interpreter and executes the corresponding programs.

The interface is patterned as a tree, in which one can move at will (fig. 2). This structure makes it possible to limit the number of commands available at each node of the tree, and these commands are displayed as menus on the screen. The menus vary at each step in the dialogue, but the commands are always very simple. If necessary the system will prompt the user and ask precisely for any complementary information required to execute a command. Incorrect input is diagnosed and will cause no error in the program, which simply goes back to the previous step.

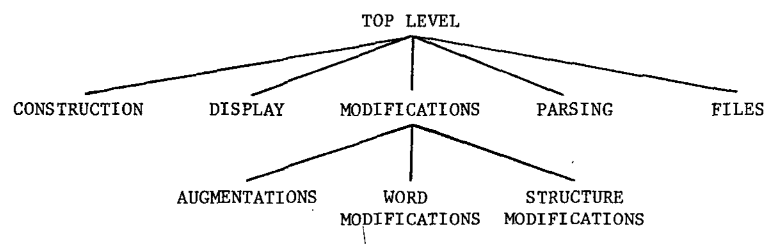


Fig. 2. Structure of the interface

We tried to classify functions in a clear way, and to split them up in short operations to avoid burdening the user's memory. Any result is displayed at once. There are never more than five or six items to consider at any moment, whether one takes into account the number of commands in a menu or the number of levels in the structure of the interface. The current situation being always indicated on the screen, there is no need to keep track of events and the system requires almost no training before use.

For example during the construction of the grammar, the branches of syntactic trees are displayed node by node while being built, so as to prompt the user and show him the current position. For each new syntactic category, ARBUS will ask for the description of one more tree until the grammar is completed. The system itself takes care of the scheduling of operations, prompts the user accordingly, and automatically builds the lexicon corresponding to the grammar defined. The user is thus guided at every step.

Automatic grapheme-to-phoneme translation of the vocabulary is also provided for speech recognition grammars. The user can input words in ordinary spelling, and they will be converted internally to phonetic form for phonemic speech recognition. Moreover pronunciation variants and linking forms are computed (work in progress by F. Néel, M. Eskénazi and J. Mariani). One may therefore define a grammar in phonetic form without any prior phonetic training and without having to do the transcription oneself.

CONCLUSION

The ARBUS system is thus a useful, pleasant and practical tool for the development of grammars. A first version was implemented in PL/I on IBM 370/168; ARBUS was then completely rewritten in INTERLISP/370, a language better suited to the manipulation of symbolic structures. Both versions are operational, but the PL/I version is directly compatible with speech processing programs written in the same language, while grammars built in INTERLISP are available through files.

We have used ARBUS to develop application grammars for speech recognition and to experiment with dialog grammars in man-machine communication. For instance it took less than half an hour to define the syntax of a spoken command language for piloting planes by voice, with about 100 words and 250 different states. This grammar was then successfully used in speech recognition. In other similar experiments we have found ARBUS pleasant to use and quite helpful as a development aid.

But it should be mentioned that this system is more appropriate for application grammars of a limited size. The deliberate choice of a tree representation for syntax and of interactive construction would make it tedious to define very big grammars in this way. For a huge syntax it would be quicker to enter it directly as a file of rules to be compiled, though the development of such a grammar would prove difficult anyway.

ARBUS might indeed be modified so as to accept rewrite rules directly. Also one could describe grammars as transition networks rather than trees. But the system would become less interactive and more cumbersome to use, while ARBUS was designed to be as interactive and as easy to use as possible. Such changes would then go against the basic purpose of the system.

Other extensions are more interesting to contemplate. When building the grammar the system could evaluate automatically the complexity of the language, according to some combination of criteria (size of the vocabulary, number of rules, branching factor, etc...). It would thus be possible to obtain meaningful comparisons between grammars to evaluate speech recognition or parsing systems. One might also better adapt ARBUS to the description of man-machine dialogs by specifying the respective roles of the user and the system in these dialogs.

In short, ARBUS is a good example of an interactive development tool, specially designed from the start to ease the user's task. Such a system is thus part of the evolution towards human engineering and graceful interaction which is becoming more and more apparent in many areas of man-machine communication.

REFERENCES

1- A. BONNET, Les grammaires sémantiques, outil puissant pour interroger les bases de données en langage naturel, RAIRO, 14(2), 1980.

2- P. HAYES & R. REDDY, An anatomy of graceful interaction in spoken and written man-machine communication, Computer Science Dept., Carnegie Mellon Univ., 1979.

3- G.G. HENDRIX, LIFER: a natural language interface facility, SIGART Newsletter, 61, 1977.

4- G.G. HENDRIX, E.D. SACERDOTI, D. SAGALOWICZ & J. SLOCUM, Developing a natural language interface to complex data, ACM Transact. on Database Systems, 1978, 3.

5- J.J. MARIANI & J.S. LIÉNARD, Eléments linguistiques et cognitifs dans un système de communication vocale homme-machine, Syntaxe et Sémantique en Compréhension de la Parole, AFCET-GALF, 1980.

6- F.C. PEREIRA & D.H.D. WARREN, Definite clause grammars for language analysis - a survey of the formalism and comparison with augmented transition networks, Artificial Intelligence, 13(3), 1980.

7- E.D. SACERDOTI, Language access to distributed data with error recovery, IJCAI, 5, 1977.

8- W.A. WOODS, Transition network grammars for language analysis, CACM, 13(10), 1970.