# A PARSER FOR A SPEECH UNDERSTANDING SYSTEM

by
William H. Paxton
and
Ann E. Robinson
Artificial Intelligence Center
Stanford Research Institute
Menlo Park, California 94025

## Abstract

This paper describes a parsing system specifically designed for spoken rather than written input. The parser is part of a project in progress at Stanford Research *Institute* to develop a computer system for understanding speech. The approach described uses as much heuristic knowledge as possible in order to minimize the demands on acoustic analysis.

## Parsing Speech

Parsing continuous spoken English is a significantly different problem than parsing written English, There are at least two major differences between spoken and textual input.

First, spoken input is not as easily decoded as text input. There are difficulties associated with establishing the boundaries of the particular words in the acoustic stream, with accounting for the variations of a word by different speakers, for different pronunciations of a word by the same speaker, and for different pronunciations of the same word in different contexts. Recognition of spoken input words is much more complex and much leas reliable than recognition of written input words.

*Second, speech provides information by* prosodic features—intonation, stress, pause, Juncture,[1,2] For example, intonation contours, the variations in pitch and rhythm in an utterance, provide clues to the location of phrase boundaries and to syntactic relationships of the words. (Work *is* now being done, particularly at the UN1VAC Corporation[3] and at the University of Michigan,*'[6] on how to extract such information from the acoustic signal and how to Incorporate it into a grammar for the computer.)

## A Strategy for Parsing Speech

In one approach to parsing text, the parser reads the input words, looks them up in the lexicon, and uses their syntactic features for guidance. This approach ie much less desirable for parsing speech because of the problem of reliably separating and identifying the input words using current acoustic techniques.

Because of this limitation of acoustic recognition, we have chosen to restrict the detailed acoustic processing to testing words that have been hypothesised by the parser on the basis of a wide range of syntactic, semantic, and prosodic knowledge. We expect the shift to verification of parser-proposed words to lead to more reliable acoustic decisions since the verification algorithms can be tailormade for the individual words and can take context into consideration. Moreover, by hypothesizing words in order of their likelihood in a particular context, we should be able to reduce the average number of incorrect words proposed and, hence, the potential for errors in acoustic recognition.

An early version of a system partially using this approach has been put together by making modifications to Terry Winograd's program.[6,7] Our new system is not directly based on Winograd's *program,* but it *reflects* his influence. It also has been influenced by the work of Bill Woods and Ron Kaplan on parsing systems.[8,9,10]

While keeping the approach mentioned above, the *new system differs from* the *old one in many* ways. Significantly, it uses a "best-first" parsing strategy in place of the more conventional "depth-first" strategy. Both strategies are designed to deal with "choice points" in the grammar—places where there are several alternatives for continuing the parse but not enough information available to decide among them. With the depth-first strategy, choice points are handled by using "backtracking." (A single path from the choice point is pursued until an inconsistency with the input is found. At that time, the path is permanently abandoned. The parser is backed up to the most recently encountered choice point to try the next alternative.)

There are several reasons why this depth-first method is unattractive for parsing speech. First, the uncertainty of acoustic recognition makes it difficult to decide conclusively whether a path should be abandoned. With speech, it makes more sense to deal with the likelihood of a path than with a categorical acceptance or rejection. A second objection is that the depth-first strategy makes it impossible to hypothesize words in order of their likelihood. Before the other alternatives at a choice point can be considered, all possibilities for satisfying the first alternative must be explored. This forces the testing of all words corresponding to the first alternative regardless of their likelihoods. A final objection to the depth-first strategy is that it explores many more false paths than a strategy using heuristic knowledge to guide the parse. Extra false paths are particularly bad for speech because of the high cost of the acoustic tests needed to determine that a path is a dead end and the danger of following false paths farther than necessary due to uncertain acoustic recognition. Research in Artificial Intelligence has shown that substantial increases in efficiency can result from the application of heuristic knowledge to guide searches. Equivalent gains should be possible if knowledge is used to guide the search for successful parse.

*See for instance NIlsson's book and the work referenced there.[11]

These considerations all support a best-first parsing strategy. In this approach, each new path resulting from a choice point is assigned a priority according to its estimated likelihood of leading to a correct parse. The paths are then added to the set of all paths that have been generated but not yet extended during this parse. The system follows the highest priority path from the comprehensive set until its priority drops or it reaches a choice point. At that time the cycle repeats. Since the new path chosen need not be one of the successors of the previous path, the parser will not necessarily continue along a single path until it reaches a dead end. Instead, it will suspend a path when there is an alternative available with a higher estimated likelihood. It will resume the original at a later time if it becomes most likely again.

The best-first method avoids the objections made to the depth-first strategy. First, it is not necessary to decide conclusively that a path is inappropriate, to force a "yes or no." The gradations of confidence in acoustic tests can be reflected in a range of priorities. Second, words can be hypothesized in order of their likelihood since there are no constraints imposed by the parsing strategy on the order in which paths are explored. And finally, the number of false paths explored can potentially be reduced by using heuristic knowledge to guide the parse.

Of course, this potential for reducing false paths can be realized only if extensive knowledge is successfully incorporated. Winograd made a noteworthy step in this direction by using semantics to filter out uninterpretable phrases as soon as they were parsed. But more can be done; knowledge can be used to guide the parsing rather than to act simply as a passive filter, and a wider range of knowledge can be used. For instance, in addition to semantics, the parser could be guided by such things as prosodies, statistics regarding vocabulary and syntactic constructs, models of the user and the dialog.

## Control Structures

The attempt to incorporate more knowledge may appear foolhardy in the light of the great complexity of existing systems. Winograd himself has remarked that his program was nearing the limits of comprehensibility; our early system was certainly no better. To combat this problem, we have introduced control structures to encourage the careful, systematic use of heuristic knowledge and aid *clarity of the parser.*

The control structures differentiate between the priority functions that embody the special knowledge and the other parser functions that embody the grammar. The grammar functions alone define the possible paths, while the heuristic functions control the order in which the paths will be explored. In this way the grammar is not obscured by the logic needed to assign priorities. The division is maintained by providing each alternative with its own priority function. This both furthers the incorporation of knowledge and simplifies the development of heuristics. We do not have to try to build an omniscient semantics module capable of evaluating any configuration; our system will use the particular semantic considerations in conjunction with other relevant information specific to a place in the grammar.

Control structures also replace GOTO's with functions that give a clear and explicit form to the standard parsing operations such as listing alternatives and identifying optional elements. We agree with Dijkstra and others who find GOTO's particularly harmful to program clarity.[12-14] Many other parsing systems have treated the program as a collection of labeled blocks of instructions with control transferred arbitrarily by explicit, reference to labels. This amounts to the unconstrained use of GOTO's whether the particular syntactic form is a multidirection branch statement, a state transition, or a production language formalism. The elimination of GOTO's is probably the most important way in which the control structures reduce the complexity of the parser.

Finally, there are advantages in developing the control structures as an extension to the LISP language. These include complete freedom in the use of procedures for structuring the system, availability of the control and data manipulation facilities of LISP, and compatibility with standard debugging and program development aids .

Before describing the control structures that have been added to LISP, it is useful to review the overall parsing strategy and introduce some new terminology. As the system attempts to understand an input, it uses the grammar to generate a sequence of paths. Corresponding to each path is a "process." The likelihood of a path is determined by priority functions and is reflected in the value of the priority for the process. Priorities are positive numbers whose values are irrelevant except to establish an order of all the processes. The highest priority process is run until cither its priority drops as *a* result of some test or it reaches a choice point. In either case, control is transferred to the highest priority member of the list of processes. Parsing continues until an acceptable interpretation of the input is found or some resource bound is exceeded.

With this strategy in mind we can discuss the actual control functions. The most basic of these is ALT, which is used to list alternatives . Its syntax *is* (ALT $alt_1$ ...$alt_n$), where each alt. is of the form {altname altpriority $e_1$ ... $e_n$). Altname is a unique name for this alternative. The use of the name is described in the section PATHS and MAPS. Altpriority is evaluated to determine the priority to be given the process corresponding to this alternative. Finally, $e_1$ ... $e_n$ specify the action for this alternative. Actions can include calling other control functions or procedures that include control functions. In other words, there is complete freedom in dynamically nesting control structures. The effect of the ALT is to replace one process by several new ones. Each new process independently can continue the computation started by the original process. When one has finished its action e. ... e , it can immediately go on to the statement following the ALT.

A second control function is OPTION, which is used to identify optional components in the grammar. Its form is (OPTION optname optpriorities $e_1$ ... $e_n$), where optname is a unique name for this option, optprlorities evaluates to a pair of priorities, and $e_1$ . . . $e_n$ specify the optional action. The first of the priorities is assigned to a process that will execute $e_1$..............$e_n$, and the second to a process that will simply go directly to the successor of the OPTION.

SEQUENCE is similar to OPTION. On entry to SEQUENCE the process Bplits into two-one which executes $e_1$..............$e_n$, and one which does not. However, unlike OPTION, the first process reenters SEQUENCE after completing $e_n$. This leads to computing another pair of priorities and spliting into one process which executes $e_1$..............$e_n$ a second time and one which does not. In this manner, the SEQUENCE statement can be reentered an arbitrary number of times to parse an arbitrarily long sequence of constituents satisfying $e_1$.............$e_n$. .

The fourth control function is OPTIONALIF. Its form is (OPTIONALIF condition name priorities $e_1$...................$e_n$). If the condition is true then $e_1$ . . . e
1 $e_n$ l n
are optional; otherwise they are required. The name and priorities for OPTIONALIF are like those for OPTION and are used only if the condition is true.

The final function is PARSE, used to call the program for a grammatical unit. Its syntax is (PARSE unit $arg_1$.......$arg_1$), where unit is the name of the function to be called for a grammatical unit and $arg_1$.......... $arg_n$ are optional arguments for that function. The result of PARSE is a parse tree for the unit. PARSE also plays an important role in the implementation as discussed below.

## An Example

Having discussed the major elements of the parser, we will now present a sample grammar for a noun phrase (Figure 1). Unfortunately, an example small enough to present here cannot fully demonstrate the value of this approach in developing a large system. But this example can Illustrate the format of a grammar. Listings of a much more complete grammar are available from the authors.

The example grammar is a LISP program called NOUNGROUP. The first statement in it is an ALT. In this example, the first alternative is

```
(ARTICLE (ARTPRESENT)
         (WDTYPE ARTICLE))
```

Altname Is "ARTICLE," altpriority is computed by the function ARTPRESENT, and the action for this alternative is the call to WDTYPE. The function WDTYPE finds in the input a word of the category specified by its argument.

In this example, $alt_1$ looks for an initial article In the noun group, $Alt_2$ through $alt_5$ look for a demonstrative adjective, quantifier, pronoun and thing pronoun respectively. The last alternative, $alt_6$ (labeled NULL) allows the noun group to start directly with an adjective or noun. The statement following the ALT controls the parsing of optional adjectives and the head noun. These are not allowed after a THINGPRON, are optional after either a DEMONSTRATIVEADJ or a QUANTIFIER, and are required in all other cases. This is represented in the program by an OPTIONALIF inside a COND. The final statement of NOUNGROUP looks for modifying relative clauses or prepositional phrases.

This grammar will find constructions such as:

the big green table"
"that part"
"some narrow pieces"
"something"
"everything green"
"he," "it"

All of these except the pronouns can be followed by a sequence of modifying phrases. Relative clauses such as "which you dropped" and "that was picked up" or prepositional phrases such as "on the floor" and "under the table" are modifying phrases.

## Word Verifier

Another important part of the parsing system is the word verifier. This component testa acoustic data for the presence of words from such terminal categories as noun, verb, and adjective that are predicted by the parser according to some path through the grammar. The words in the terminal category can be thought of as alternatives, exactly like the higher syntactic alternatives in the grammar, that should be considered in order of their likelihood. The same structure of alternatives and priorities is here found on the level of word verification. Estimated likelihoods can not only order the words in the terminal category, but also defer acoustic processing as long as higher priority alternatives remain to be explored.

The word verifier must first establish a priority for each candidate word and then, according to those priorities, schedule acoustic verification. Corresponding to these two operations there is a priority function and a verification function associated with each word in the vocabulary. The tests made by the priority function are computationally simple, although broad in scope. The results of preliminary acoustic processing are consulted to ensure that the word is at least feasible. The frequency of occurrence of the word and the likelihood of co-occurrence of the word with recently used words are considered. Finally, the semantic features of the word are compared to the semantic restrictions of the context.

In contrast to the quick tests made by the priority function, the verification function may perform extensive acoustic analyses to determine how well the proposed word matches the next portion of input. The details of this processing are beyond the scope of this paper but have been sketched elsewhere.[7] Since the verification function depends only on acoustic information, its results are saved so that, if another process looks for the same word in the same place, the prior results can be used. Of course, the results of the priority functions cannot be reused In this manner since they are extremely dependent on context.

```
(NOUNGROUP
  (LAMBDA NIL
    (PROG NIL
        (ALT (ARTICLE (ARTPRESENT)
                      (WDTYPE ARTICLE)         (* A AN THE)
                      )
             (DEMONSTRATIVEADJ (DEMADJPRESENT)
                               (WDTYPE DEMONSTRATIVEADJ)
                                               (* THIS THAT THESE
                                               THOSE)
                               )
             (QUANTIFIER (QNTFRPRESENT)
                         (WDTYPE QUANTIFIER)
                                               (* SOME NONE NO NEITHER
                                               MOST MANY EVERY EITHER
                                               EACH BOTH ANY ALL)
                         )
             (PRONOUN (PRONPRESENT)
                      (WDTYPE PRONOUN)
                      (RETURN))
             (THINGPRONOUN (TPRONPRESENT)
                           (WDTYPE THINGPRONOUN)
                                               (* SOMETHING NOTHING
                                               EVERYTHING ANYTHING)
                           (OPTION THINGADJ (ADJAFTERTPRON)
                                   (WDTYPE ADJECTIVE))
                                               (* 'SOMETHING RED')
                           )
             (NULL (NOART)                     (* NOUNGROUP STARTS WITH
                                               ADJECTIVE OR NOUN)
                   ))
        (COND
          (THINGPRONOUN NIL                    (* ADJECTIVE-NOUN NOT
                                               ALLOWED AFTER
                                               THINGPRONOUN))
          (T (OPTIONALIF (OR DEMONSTRATIVEADJ QUANTIFIER)
                         ADJNOUN
                         (ADJECTIVENOUNPRESENT)
                                               (* AFTER DEMONSTRATIVE
                                               ADJECTIVES AND
                                               QUANTIFIERS
                                               ADJECTIVE-NOUN ARE
                                               OPTIONAL)
                         (ADJNOUN)             (* PARSES OPTIONAL
                                               ADJECTIVES AND HEAD
                                               NOUN)
                         )))
        (SEQUENCE ENDINGS (ENDING)
                  (ALT (CLAUSE (CLAUSEAFTER)
                               (PARSE CLAUSE RELATIVE))
                       (PREPOSITION (PREPAFTER)
                                    (PARSE PREPGROUP)))))))
```

Figure 1

PATHS and MAPS

In the word verifier, we have seen how one process can use information gained by another; the results of a verification function can be used by a subsequent process looking for the game word. The next step is to share the syntactic structures that are found. But the processes may have had different contextual constraints and may need to make different tests, problems for direct sharing. We have added PATHS and MAPS to the parsing system to facilitate sharing and to allow for additional tests.

A path describes the flow of control through the grammar leading to the current state of a process. For example, at ALT statements the name of the alternative taken is added to the path, and when a word is recognized in the input It also is added. The actual structure of the path is a list of names and words with the most recent history of the process at the front of the list. A map is the part of the path that was followed by a process when it successfully parsed some grammatical unit. The function PARSE automatically records the map before it returns, and it checks for a map when it is called.

This Is closely related to the well formed substring facility described by Woods.[8]

The control functions (ALT, OPTION, OPTIONALIF, and SEQUENCE) consult the map and raise the priority of the named alternative. The map thus helps to guide the process which is parsing a constituent by contributing data on alternatives successful in prior attempts.

Nevertheless, it is still possible that the map will fail due to differences in the context. When a process is forced off the path specified by a map, it simply reverts to the standard mode of parsing. This ensures that the use of maps will not result in the acceptance of a previously parsed constituent that falls to meet the current constraints. The map merely serves as heuristic knowledge to guide the reparse and can be overridden by other considerations.

Finally, the map must take into account the fact that there may be more than one successful parse of, say, a noun group starting at a particular location in the input. In its most general form a map is a tree structure with each path through the tree corresponding to a successful parse. If a branch point is reached while following a map, the several alternatives will be given raised priorities.

It is worth noting a second use for the path as a debugging tool. The parsing strategy causes control to move among a large number of active processes, and the computation leading to the current state of a process will have been interlaced with the activity of other processes. Standard tracing techniques are less useful for debugging, and the path is a valuable aid in determining how a process has reached its current state.

## Implementation

The parser will be implemented in BBN-LISP using the multiple environment control structure facility described by Bobrow and Wegbreit,[16] Among other things, this facility generalizes the standard linear stack to a tree structure. Figure 2 introduces some terminology for discussing such a tree.

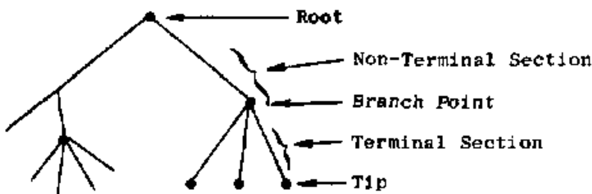
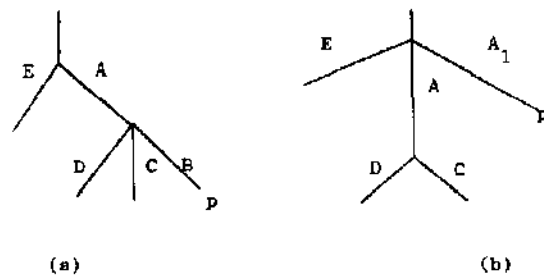



Figure 2

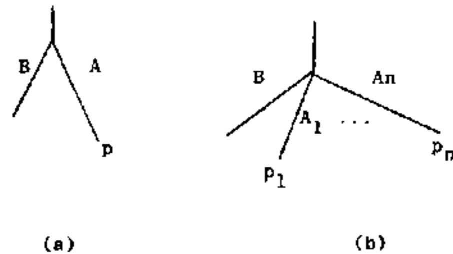
The root corresponds to the base of an ordinary stack, and each tip behaves like the top of an ordinary stack. In the parser there is a one-to-one correspondence between the tips of the stack structure and processes. When a process calls a function, the variables for the function are added to the tip for that process, and when the function returns, the variables are removed. The path from the root to the tip contains all the variables for the process. At branch points the paths for several processes join. Thus the variables in the path from the root to the branch point are shared by more than one process. Only the variables in terminal sections can be modified by the process without interfering with other processes; the variables in nonterminal sections must not be changed. This presents a problem when the process wishes to return from the first (nearest the root) function in the terminal section to the last function in the last nonterminal section. The process cannot use that section because it is shared. The solution is to make a copy of it that will have the same relation to other sections of the path as the shared section but will be for private use of the process. To ensure that this copying is done properly, each stack section begins with a call of PARSE, and instead of returning in the normal manner, PARSE makes a copy of the next section of the tree and transfers control to it. This provides a new terminal section for the process and returns control to the function that called PARSE originally. Figure 3 shows the effect on the stack structure of such a return by PARSE .




In (a), process p is running in terminal stack section B. In (b), p has "returned" from B to $A_1$, a copy of A, and B has been deleted.

Figure 3

Copying is also necessary when new processes are created. Figure 4 shows a portion of the stack before and after an ALT statement.* When a process is killed its terminal stack section is deleted.




In (a), p, running in Section A, is about to reach an ALT statement. In (b), p has been replaced by processes $P_1$ through pn, each with a copy of A, corresponding to the n alternatives of the ALT.

Figure 4

*This is not quite true. In the actual system the copy $A_1$ is not made until $p_1$ becomes the highest priority process. Thus all the new processes which have not yet been activated share the same terminal section.

There are two main restrictions on the parser caused by this implementation. First, data structures which are not intended to be global to all processes cannot be changed except after copying. This is to avoid interference between processes which may be sharing the same structures. For example, if process p in Figure 4a, has a variable X'whose value is a list, then processes $p_1$ through pn in Figure 4b will all have variables pointing to the same list (in LISP terminology, the *X's* will be EQ). To change the value of X, process $p_i$ must store a new pointer in X rather than modify the original structure pointed to by X. While this restriction has affected the design of the parser, it has not been a serious problem.

The second main restriction on the parser, that only variables within the terminal section of the stack can be changed, did present a problem. Certain variables such as the current position in the input are private to each process but global within the process. The solution is to identify such variables to the system as "process globals." On entry to PARSE, the process globals are rebound to their old values. Thus they are within the terminal section and can be given new values. Before PARSE exits, it propagates the process globals back up the stack. For example, in going from Figure 3a to Figure 3b, the process globals for p are propagated from section B to section A,. By generalizing this scheme, variables can appear to be bound at any level. Thus we can eliminate the second restriction on the parser by identifying certain special variables to the system.

The implementation has the prime advantage of avoiding a special interpreter for the parsing language, Since the "language" consists of additional LISP functions, the entire parser can be compiled with the standard LISP compiler and debugged with the standard LISP debugger. This means faster execution and easier debugging. Of course, the biggest improvement in efficiency relies on knowledge to guide the parser, but compilation will give a substantial speed up in addition.

## Conclusion

The system described in this paper is part of an. ongoing research project in speech understanding and is intended to provide a solid basis for continuing work. There are many other aspects to speech understanding besides the development of a parser system. It has been to our benefit to be part of a speech project at SRI that includes personnel interested in a variety of tasks from acoustic signal processing through semantic representation to overall system organization,

A sizeable grammar for English has been written using this parsing system. Although it is difficult to characterize the scope of a grammar, it appears to be as extensive as others we are familiar with (in particular those of Winograd[6] and Woods, et al.[9]). Our current efforts center on acoustic processing routines for use in word verification, priority functions for grammatical alternatives, and semantic modeling of the domain of discourse.

These functions of course depend on the multiple environment control facility in BBN-LISP.


## Acknowledgements

We would like to thank the people in the project for helping us to see some of the problems involved in speech understanding and to develop the approach to parsing speech described in this paper. Project leader Don Walker has been particularly helpful in this respect. We have also benefited from discussions with other participants in the Advanced Projects Agency speech understanding program.[17] The research reported herein was supported at SRI by the Advanced Research Projects Agency of the Department of Defense, monitored by the U.S. Army Research Office-Durham under Contract DAHC04 72 C 0009.


## References


1. 1. Lehiste, Suprasegmentals, MIT Press, Cambridge, Massachusetts (1970).
2. P. Lieberman, Intonation, Perception, and Language, MIT Press, Cambridge, Massachusetts (1967).
3. W. A. Lea, H. F. Medress, T. E. Skinner, "Prosodic Aids to Speech Recognition," Technical Reports PX 7940 and PX 10232, UNIVAC Corporation, St. Paul, Minnesota (October 1972, April, 1973).
4. M. H. O'Malley, "The Use of Prosodic Units in Syntactic Decoding," presented at the Session on Linguistic Units at the 85th Meeting of the Acoustical Society of America, Boston, 13 April 1973 (University of Michigan, Ann Arbor, Michigan (1973)).
5. J.J. Robinson, "predicting Phonological Phrases," Manuscript from the University of Michigan, Ann Arbor, Michigan (1973).
6. T. Winograd, Understanding Natural Language, Academic Press, New York, New York (1972).
7. D. E. Walker, "Speech Understanding Research," Annual Technical Report, Contract DAHCO4-72-C-O09, SRI Project 1526, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California (February 1973).
8. w. A. Woods, "An Experimental Parsing System for Transition Network Grammars," in Natural Language Processing, R. Rustin ed., Algorithmics Press, New York, New York, pp. 111-154 (1973).
9. W. A. Woods, R. M. Kaplan, B. Nash-Webber, "The Lunar Sciences Natural Languages Information System: Final Report," Report No. 2378, Bolt Beranek and Newman, Inc., Cambridge, Massachusetts (1972).
10. R. M. Kaplan, "A General Syntactic Processor," in Natural Language Processing, R. Rustin ed., Algorithmic Press, New York, New York, pp. 293-241 (1973).
11. N. J, NIlsson, Problem solving Methods in Artificial Intelligence, McGraw-Hill, New York, New York (1971)
12. E. W. Dijkstra, "GOTO statement Considered Harmful," letter to editor, CACM, Vol. 11, No, 3, (1968).
13. W. A. Wolf, "Programming Without the GOTO," Proc. of IFIP Congress (1971).
14. 0. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press, New York, New York (1972).

15. W. Teitelman, et al., BBN-LISP, Bolt Beranek and Newman, Inc., Cambridge, Massachusetts (1971).
16. D. G. Bobrow, B. Wegbrelt, "A Model and Stack Implementation of Multiple Environments," Report No. 2334, Bolt Beranek and Newman, Inc., Cambridge, Massachusetts (1972).
17. A. Newell, et al., Speech Understanding Systems, North-Holland Publishing Company, Amsterdam, The Netherlands (1973).