AI Magazine Volume 8 Number 1 (1987) (© AAAI)

John C. Glasgow II

YANLI: A Powerful Natural Language Front-End Tool

An important issue in achieving acceptance of computer systems used by the nonprogramming community is the ability to communicate with these systems in natural language. Often, a great deal of time in the design of any such system is devoted to the natural language front end. An obvious way to simplify this task is to provide a portable natural language front-end tool or facility that is sophisticated enough to allow for a reasonable variety of input; allows modification; and, yet, is easy to use. This paper describes such a tool that is based on augmented transition networks (ATNs). It allows for user input to be in sentence or nonsentence form or both, provides a detailed parse tree that the user can access, and also provides the facility to generate responses and save information. The system provides a set of ATNs or allows the user to construct ATNs using system utilities. The system is written in Franz Lisp and was developed on a DEC VAX 11/780 running the ULTRIX-32 operating system.

Augmented Transition Networks

The system is named YANLI in (shameless) imitation of "yet another compiler compiler" (YACC) (Johnson 1984). The programs are similar in that at the heart of both programs is a parser whose grammar is definable and serves as the "source code" for the parser.

Additionally, both programs can perform actions at any step during the parsing of input. However, YACC uses a bottom-up (left-to-right) parsing method and is comfortable with the grammars that conveniently describe programming languages, and YANLI has a top-down parser based on ATNs and is intended to parse a subset of common English. YANLI is actually a set of tools for the development of ATNs together with several built-in ATNs that describe a subset of English.

John C. Glasgow II is systems manager for Florida Agricultural and Mechanical University and Florida State University (FSU) College of Engineering, Department of Computer Science, Tallahassee, Florida 32306 Mr Glasgow is also a Ph D. candidate at FSU, Department of Computer Science. For information about obtaining YANLI write to: John Glasgow, c/o FSU, Department of Computer Science, room 203, Love Building, Tallahassee, FL 32306

Since first proposed by Woods (1970), ATNs have become a popular method of defining grammars for natural language parsing programs. This popularity is due to the flexibility with which ATNs can be applied to hierarchically describable phenomenon and to their power, which is that of a universal Turing machine. As discussed later, a grammar in the form of an ATN (written in LISP) is more informative than a grammar in Backus-Naur notation. As an example, the first few lines of a grammar for an English sentence written in Backus-Naur notation might look like the following:

```
sentence  ::= [subject] predicate
subject   ::= noun-group
predicate ::= verb [predicate-object]
    .              .
    .              .
    .              .
```

Figure 1 illustrates this same portion of a grammar when it is written as transition nets (TNs).

These TNs (ignoring augmentations for the moment) have more of a procedural presentation than the same grammar in Backus-Naur notation. Because words such as "if" and "transit-to-state" are included in the TNs, the process that must take place in recognizing a sentence of input is easily discerned. The Backus-Naur notation makes only a static definition of the grammatical structure. Actually, the words "if" and "transit-to-state" in the TNs are just space fillers, and the "test-for-xxx" words (referred to here as "states") serve only to mark the right hand of each production rule in the Backus-Naur form. Thus, no procedural

Abstract This article describes "yet another natural language interface" (YANLI), a small portable natural language interface based on augmented transition networks The system consists of a parser, a set of augmented transition nets that represent a subset of English, utilities to allow modifications to the system, and utilities to access the output of the system. Included is a response generator that allows the system to be used "as is." Also presented is an example of how the system can be applied.

```
(sentence
   (test-for-subject
      (if subject transit-to-state test-for-predicate)
      (if t transit-to-state test-for-predicate))
   (test-for-predicate
      (if predicate transit-to-state win))
      (if t lose)))
(subject
   (test-for-noun-group
      (if noun-group transit-to-state win))
      (if t lose))
(predicate
   (test-for-verb
      (if verb transit-to-state
       test-for-predicate-object))
      (if t lose))
   (test-for-predicate-object
      (if . . .
```

Figure 1. Example of a Grammar in the Form of a Transition Network.

meaning is attached to these TNs that is not implicit in the Backus-Naur grammar; one can be converted directly into the other. Still, the form of TNs assists a person with the construction of a grammar by making it easier to conceptualize the process that is to take place. The process implied by the form and content of TNs is carried out by the parser.

The Parser

The parser works by traversing the TN list and making transitions as appropriate. For instance, the parser evaluates the second argument in a list headed by an "if" as a new TN to be traversed; so, in figure 1, the words subject and predicate following the "if's indicate to the parser that it should now traverse the TN of that name. Proceeding in this manner, the parser must eventually arrive at a TN that tests a word of input. If the word exhibits the proper feature (for example, is a noun where a noun is required, which is information found in a dictionary provided by YANLI), then the parser pops back to the previous TN reporting success (win). However, if the word does not pass the test, the parser pops back reporting failure (lose). Now, the parser continues in the manner implied by the if. In the event of a win, the traverse proceeds to whichever state is indicated after the transit-to-state word of the if statement. In the event of a lose, the parser proceeds to the next if list in the TN. If there are no more if lists, it pops back again reporting "lose." In any given TN, the parser must eventually reach the win or lose state or run out of if lists. Inevitably, the parser must return to the TN from which it began in either a win or a lose condition. The win indicates that the words of input seen during the traverse were a sentence of the grammar, and a lose means the words were not a sentence of the grammar. Thus, what YANLI recognizes as acceptable, is defined by the grammar represented by TNs. A sentence of the grammar need not be a correct English sentence, but simply a structure that should be recognized. For example, a tutoring system might ask, "Do you understand the problem?" and the student might respond, "not really." Although the response is not a sentence, the system would expect such a response, and the grammar would be defined to accept it. The English recognizing networks supplied with YANLI are similar in appearance to the ATN shown in figure 2. To make the process that is to occur as explicit as possible, notations such as (np/ (cat det . . .)), (jump np/ . . .), and so on, have been dropped in favor of descriptive phrases.

The parser, operating on TNs, is capable only of recognizing a sentence of the grammar. To be useful, TNs must be augmented with the ability to perform actions. Providing TNs with this ability requires the addition of some extra elements to the TN lists. These extra elements will be interpreted as Lisp symbolic expressions, or *s-expressions*. S-expressions are evaluated when and if the parser encounters them in the traversal of ATNs (see figure 2). When evaluated, an s-expression always returns a value (although the main purpose might be to cause some side effect).

```
(sentence
   (test-for-subject
      (if subject transit-to-state test-for-predicate)
      (if t transit-to-state test-for-predicate
         but-first
         (addr 'comments 'no-subject))
   (test-for-predicate
      (if predicate transit-to-state win
         but-first
         (print ''It's a sentence''))
      (if t transit-to-state lose))))
```

Figure 2. An Example of an Augmented Transition Network.

The addition of the element "but-first" to an if list signals the parser to look for s-expressions to execute. In the figure, a return to the previous ATN from the test-for-predicate state (win) results in "It's a sentence" being flashed on the screen, and, significantly, if the parser had failed to find a subject in "test-for-subject," then "no-subject" would have been added to the property list of "sentence" under the register "comments." At a later time, this register can be accessed, and it can be deduced that the sentence is a command or a question. It is useful to be able to issue messages to the screen during a parse, but it is more

. . . A problem in natural language processing is determining when the time is appropriate to perform syntactic and semantic processing . . .

important to be able to deposit messages at different times during the parse to be read at a later point in the parse or after the parse has concluded. The augmentation makes the parsing system powerful enough to do anything that can be done by high-order (third generation) programming languages.

As the parser proceeds through ATNs, it constructs a tree by joining, as limbs, those subtrees from which it returns in the win state. Upon returning to the starting ATN, it has constructed a tree of nodes that represents the structure of the input sentence it has just read. This *parse tree* is the output of the parser and contains all the information that the parser has been able to gather about the sentence by matching it to the grammar represented by ATNs. The tree structure consists of nodes that contain named registers. The registers can contain any information about the input that the parser, following the pattern of the ATNs, was able to deduce. The content of the parse tree thus created depends completely upon how carefully and for what purpose the ATNs were constructed. The purpose may be as simple as recognizing a sentence of the grammar or as difficult as understanding the semantics of that sentence. The purpose of YANLI's ATNs is to discover the syntactic structure of the input sentences (as well as it can without using world knowledge) and to make the elements of that structure easily accessible to other programs, in particular to make these elements available to the response generator.

A problem in natural language processing is determining when the time is appropriate to perform syntactic and semantic processing: Should the syntactic processing occur first with a separate semantic parse later, or should both kinds of parsing be done at the same time? Arguments (and parsers) can be made for both sides. It would be impossible to write a complete grammar for common English without providing for semantic analysis. For instance, the phrase "have the doors closed" can be interpreted either as the command "You have the doors closed" or as the question "Are the doors closed?" The meaning of the phrase, which must be determined from the context in which the phrase is issued, is crucial to the way it is parsed. Thus, the requirement exists that knowledge of the situations in which this phrase might occur, be available to ensure a correct syntactic parse. However, building a grammar for common English and a parser which can take advantage of world knowledge so that ambiguities can easily be handled is difficult, and the result of such a project would be an unwieldy and impractical natural language front end (at least on existing machines).

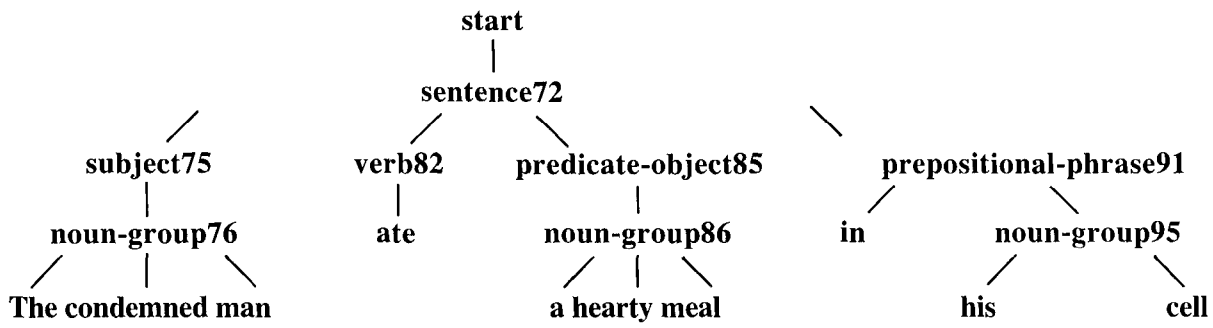
Fortunately, for every ambiguous statement, there are many semantically equivalent statements that are not ambiguous. By restricting conversations with computers to unambiguous or easily disambiguated and simply structured sentences, useful grammars which do not rely on world knowledge and which are comfortable for people to use can be built. This flexibility of English is good because for a small natural language interface such as YANLI, space and time considerations require that the parser conduct a mostly syntactic parse. YANLI's built-in ATNs contain very little knowledge that does not pertain to the syntax of English grammar (however, there is no reason why semantic knowledge cannot be added as needed). When YANLI is used with its ATNs unmodified and without the response generator, an analysis of the parse tree generated during a parse is necessary to be useful.

Utilities

ATNs provided with YANLI cause a number of items to be stored in registers attached to the nodes of the parse tree; these registers serve to direct the parse and to capture the syntactic structure of the input. These registers include items such as children nodes, parent nodes, parts of speech, parsing data, input words, clause sequence, sentential constructs, number values, and comments. Although these registers save sufficient data for many analytical purposes, other things, especially semantic information, might be usefully included for particular purposes. The parser utilities such as addr, setr, setf, getr, getf, movr, and peek (look ahead) which are common in the literature (see, for example, Christaller 1982; Finin 1977; Harris 1985) and which provide the ability to add registers and data to nodes and to test the data and the input are available in YANLI so that the ATNs supplied with it can be modified to provide desired additional data collecting and testing. Furthermore, YANLI's ATNs can be completely replaced or used as part of other ATNs that conform to the YANLI ATN structure.

To facilitate the construction of ATNs in a form amenable to YANLI's parser, a program is provided as a guide. The program is activated by "(make-atn)" and presents a template of an ATN so that it is only necessary to fill in the blanks. The output is an ATN directly usable by YANLI. To aid in debugging, YANLI can be run in a mode that displays every step in its parsing process.

To provide access to the parse tree, two display functions, two access functions, and a storage-retrieval facility are provided. The access functions are "poll" and "extract." Both functions traverse the parse tree generated by YANLI gathering information. During the parsing process, YANLI stores information in the registers of the nodes of the parse tree. This information, as indicated earlier, can range from information about the part of speech of a word of input to the sentential structure of the entire input. Poll looks for the contents of a specific register from a specific node on the tree and extract gathers the contents of a specific register type along a given branch of the parse tree. For instance, the parse tree for the sentence "The condemned man ate a hearty



Note: The digits attached to the parts of speech are appended by YANLI and make each node of the parse tree unique.

Figure 3. Example of a Parse Tree.

meal in his cell" could be represented graphically as shown in figure 3.

In words, the function call

```
(extract 'word 'subject 'start)
```

means extract the contents of the register 'word from every node of the subtrees whose root nodes are of type 'subject and do this for all the subtrees in the tree whose root name is 'start. For figure 3, the list "(The condemned man)" is returned. The function call

```
(poll 'word 'subject 'start)
```

means extract the contents of the register 'word from the nodes of type 'subject and do this for all the nodes in the tree whose root node is 'start. In this case, because YANLI does not attach a register for words to nodes of type 'subject, no words are returned. By combining these functions, any part of the parse tree can be accessed; for example:

```
(extract 'word

(car (poll 'noun-group 'predicate-object 'start))
'start)
```

returns "(a hearty meal)," because the embedded function call

```
(poll 'noun-group 'predicate-object 'start)
```

returns "(noun-group86)" as the noun group in the noun-group register attached to the predicate-object node. Then,

```
(car '(noun-group86))
```

evaluates to "noun-group86" so that the final evaluation is of

```
(extract 'word 'noun-group86 'start) .
```

This evaluation returns a list of all the words in noun-group86, that is, "(a hearty meal)."

Because YANLI is designed to run on standard DEC terminals, a graphic representation similar to that shown in figure 3 is difficult to produce and wastes screen space. So that the parse tree can be displayed, the function "write-parse" is provided. It is called with one argument that represents the node at which to begin, for example, "(write-parse 'start)," and displays the parse tree on the screen in an outline form (see figure 4) that is equivalent to the tree in figure 3.

```
start                                nil
   sentence..00072                   nil
      subject..00075                 nil
         noun-group-type-1..00076    nil
            the                      nil
            condemned                nil
            man                      nil
      verb-group..00082              nil
         ate                         nil
      predicate-object..00085        nil
         noun-group-type-1..00086    nil
            a                        nil
            hearty                   nil
            meal                     nil
      prepositional-phrase..00091    nil
         in                          nil
         noun-group..00095           nil
            his                      nil
            cell                     nil
```

Note: Comments are written at the right when found attached to the node; otherwise, "nil" is written

Figure 4. Display of a Parse Tree in Outline Form.

Activating YANLI

The functions "verbose" and "talk" activate YANLI in a mode that causes the parse tree, its associated node list, and all the values attached to these nodes to be stored in a file. Stored data can be retrieved later using the function "(retrieve filename)," where "filename" is normally the default storage name "parsout.dat". The function verbose causes

each step of the parse to be displayed, and talk issues occasional prompts for input but otherwise remains silent. Another function, "silent," can be used to embed YANLI in an application; it returns nothing to the screen and stores no information, but it does generate the parse tree that can be accessed using poll and extract. Each of the calls can be made with an optional argument, "atn," where atn is the name of a foreign ATN. For instance, "(verbose 'my-atn)" activates YANLI specifying that "my-atn" is to be used in place of ATNs provided with YANLI.

Two other modes of YANLI activation are "respond" and "build-responses." The call build-responses allows the creation of a response and the association of this response to the input if no response has already been associated with the input. It prompts for input and provides several displays to guide the construction process. The function respond waits silently for input from the terminal, parses this input, and makes a response if one has been provided or makes a default response. The function respond is equivalent to silent in that it can be used as an embedding mechanism for applications which use the response-generation feature of YANLI. The final modes for activating YANLI are "(respond-to input [atn])," "(talk-to input t/nil [atn])," and "(silent-to input [atn])." These modes are the same as the "to" less version of the same function except that input is accepted from a program (the "input" argument) instead of the keyboard.

Preprocessor

The input to YANLI is processed before being passed to the parser. During this preprocessing phase, all capital letters are converted to lowercase letters, sentence punctuation is recognized and replaced by markers attached to the property lists of the words ending sentences, contractions and abbreviations are replaced by their full-word equivalents, numbers are replaced by variable names to which the numeric value is attached as a register (so that number values can be treated as data in the same manner as word features), some sentences with ellipses are made whole, and short idiomatic expressions or terse ungrammatical responses are converted to full sentences with equivalent meaning. The conjunctions, abbreviations, and expression equivalents are maintained in a table separate from the compiled YANLI code so that additions can be made to the set of entries provided. During preprocessing, if a word is detected that is not in the dictionary, the user is prompted for a definition, which can, if desired, be made a permanent entry in the dictionary.

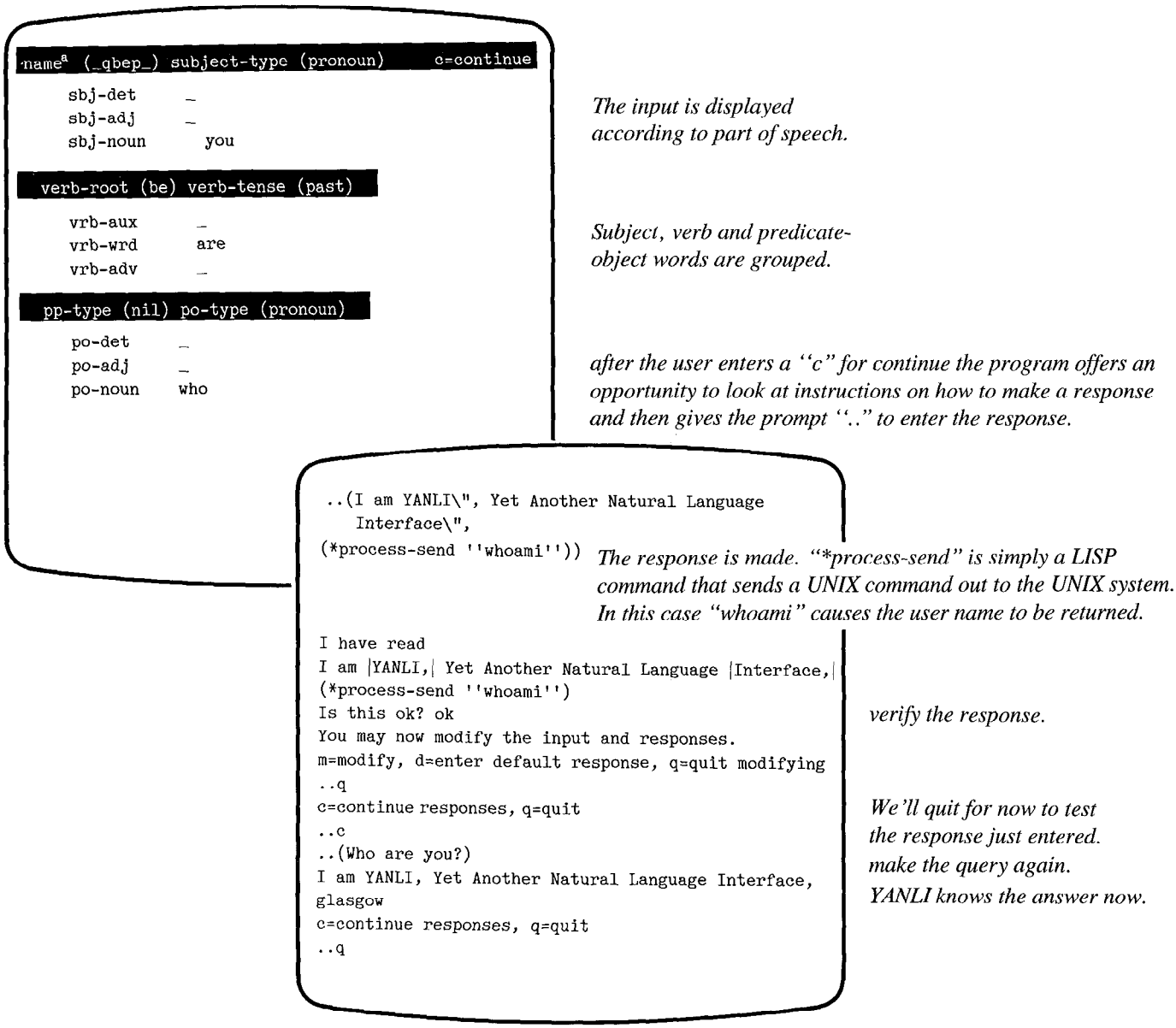
The Response Generator

The parser and associated tools have been used to construct a response generator so that YANLI can be used as both the input and the output interface for application programs. The action of the response generator is to produce a pattern from a given input sentence and allow this pattern to be modified as a template for matching other similarly formed sentential inputs. A hand-built response can then be associated with the template so that it is invoked whenever a sentence which matches the pattern is detected. The response can be constructed to depend upon the elements which fill in the slots in the template so that it is customized to the particular words of the input. Furthermore, the response can be executable s-expressions or a combination of text and s-expressions, and the s-expressions can contain the elements that fill in the slots in the template. Each template and the responses it contains are stored under a unique name that describes the structure of the template. This storage facility creates a file system, each file of which matches a generic kind of sentential input and contains the correct responses for variations on these kinds of input. These response files need not be loaded into memory until the parser detects a sentence of this type. For a response generator of this kind to be effective, YANLI need only be able to create unique patterns for the sentences that are input to it. Knowledge contributing to the construction of a correct response comes from the person building the response or a program supplied by that person. The response generator works as shown in figure 5. This figure illustrates that a possible application of YANLI is as a natural language interface for UNIX.

1) YANLI is activated with the command "(build-responses)".
2) YANLI waits for input, reads it and parses it.
3) If there is already a response for that particular input then the program just makes the response and goes to 2.
4) YANLI displays a table of most of the words of the input classified as to sentence structure and part of speech.
5) YANLI requests an appropriate response for this input. A response may be constructed that uses the parts of speech as variables or that contains executable LISP expressions. You may quit.
6) YANLI displays the table again and asks for modifications to the original input. These modifications may include wildcards, indicating that any word should be accepted, or a list of alternative acceptable words. YANLI goes to 5.

For example:

```
→(build-responses)                 comments are in italic
                                   "→" is the Franz Lisp prompt
..(who are you?)                   ".." is the YANLI prompt
                                   start YANLI
                                   we will build an answer to
                                   this query. YANLI parses and
                                   presents a table. This is only
                                   part of the 21 categories that
                                   would normally be displayed
```



[a]Lines displayed in reverse video head categories.

Note: Naming conventions are discussed in the subsection Classification of Input

Figure 5. Example of How the Response Generator Works.

An Application of YANLI

One recent application of YANLI is as a parser of arithmetic word problems. These problems are to be included in a problem bank that is part of an arithmetic word problem-solving tutorial system. The system is being developed by the Training Arithmetic Problem-Solving Skills (TAPS) project under the direction of Dr. Lois Hawkes and Dr. Sharon Derry at Florida State University. Part of the project requires that the tutoring system be able to solve arithmetic word problems by recognizing each problem as one or more schemas for which a solution method is known by the system. The aim is to help the student recognize the schemas and learn how to solve the problems with the given methods. YANLI's part in the system is to parse the arithmetic word problems and compile the information in the parse tree into a form that can be used by a semantic analyzer (YANLI will also play a role in the interface with the student).

The semantic analyzer determines the schema to which the problem belongs, and this information along with values from the parse tree is then given to a problem solver. The information required by the semantic analyzer is well specified so that all YANLI must do is extract the information and present it in a list. Although this process could be performed interactively during a tutoring session, the intention is to provide the means of entering new problems in preprocessed and readily accessible form into the system's problem bank to be drawn upon during tutoring sessions.

The system is aimed at grade school children and poses problems such as the following (which is chosen from the simplest variety): Joe had 3 marbles. Tom had 5 marbles. Then, Joe gave Tom his marbles. How many marbles does Tom have now?

YANLI is capable of parsing multiple sentences and creates a register named "clause-sequence" into which a list of the types of the sentences are put. For this set of sentences, the register and contents look like the following:

```
( . . . clause-sequence
(declarative declarative declarative
                          how-much) . . . ) .
```

This register provides an obvious signature for a word problem (which would be important if YANLI were to play an active part in the system). The parse tree is converted by the response generator into the pattern shown in figure 6 (omitting categories into which no words fall for the sake of brevity).

Figure 6. Example of a Pattern Resulting from Conversion of a Parse Tree by the Response Generator.

The data required by the semantic analyzer are specified for each sentence in the problem. The information required from sentence 1 is the following:

```
(((verb-tense verb-root)
 (subject-type subject-noun)
 (predicate-object-noun
   predicate-object-quantity)) .
```

The second and third sentences supply much the same information, but the last sentence requires the following:

```
((question-type)
 (predicate-object-noun)
 (subject-type subject-noun)
 (verb-tense verb-root))) .
```

All the elements of a pattern can be used in the response either individually or as a group. For instance, by including "sbj-noun" in the response, the textual list "joe tom joe tom" is produced at the point of substitution; "(sbj-noun)" produces the Lisp list "(joe tom joe tom)"; and "(sbj-noun 3)" produces the third noun, or "joe." The items of information other than parts of speech can also be accessed; for instance, the tense of the second verb can be selected using "(verb-tense 2)," which produces "past." Thus, a response consisting of any information gathered during the parse, even one that uses the information in s-expressions, can be constructed. An appropriate response to the example in figure 6 is to set a previously created global variable, say "data," to the list of lists. The constructed response, in part, is the following:

```
((setq data (list (list (verb-tense 1)(verb-root 1))
     (list (sbj-type 1)(sbj-noun 1))
     (list (po-noun 1)(po-rl-val 1))) . . .
 (list '(how-many) (predicate-object-noun)
     (list (sbj-type 4)(sbj-noun 4))
     (list (verb-tense 4)(verb-root 4)))),
```

which, when activated as a response causes the value of data to become the following:

```
(((past have)(name joe)(marbles 3))
                          .
                          .
                          .
((how-many)(marbles)(name tom)(present have))) .
```

YANLI allows you to modify the input and provide a response for the modified input. If a wild card value were substituted for "joe," "tom," "marbles," "3," and "5" and if the response were kept the same, then many equivalent but different-sounding problems could be parsed correctly for the semantic analyzer.

Classification of Input

The name shown in the displays in the figures is generated by classifying the nouns and verbs of the sentences that are input to YANLI. The intention is to partition possible inputs into named generic categories so that responses to similar sentential structures can be stored together in secondary memory. These responses need not be loaded into random-access memory until required. The information about each sentence of the input (see figure 7) is used in the classification scheme.

1) context	(supplied by the user)
2) type sentence	(d=declarative, q=interrogative, i=imperative,m=multiple sentences)
3) subject type	(n=name, p=pronoun, o=object)
4) verb-root	(the actual root form of the verb)
5) prepositional-phrase-type	(n=name, p=pronoun, o=object)
6) predicate-object-type	(n=name, p=pronoun, o=object)

Figure 7. Sentential Information Used in the Classification Scheme.

If there is no information for a category, then an underscore character is used in place of the character that is otherwise supplied by the scheme. When there are multiple sentences or multiple items in a category, each item contributes a character (or word in the case of context or verbs) and is separated by a "+" from the other items in the category. In the tutoring example, the fact that no context was supplied forces the first character in the name to be the underscore character. The type sentence is "m" for multiple sentences. The subject type is "(name name name name)," which produces "n+n+n+n" in the next part of the name. The other items in the classification list contribute their part so that we finally derive the name

'_mn+n+n+nhave+give+do+haveno+o+o+o+o'.

For ambiguous sentences or phrases such as "have the doors closed," the device can be useful. The main purpose, though, of the context category is to allow for different responses in different situations, even when there is no mistake about meaning. Such an instance can occur, for example, when a student (or computer user) asks for help. The help given by a computer program should depend upon the situation. YANLI initializes a global variable named context to value "_". This variable can be changed at any time by the user's program so that the current context is reflected in responses from YANLI. Assume that there exists a help function, "(help x)," which accesses different help libraries named in its argument x. An example illustrating the construction of a help function is given in figure 8.

If the application program keeps track of the situation by setting the context variable to "sit-a" or "sit-b," the correct help is given whenever "(help)" is input. If the context is set to neither sit-a nor sit-b, then a default response is generated.

Default Responses

When an input to YANLI is not recognized by the grammar, YANLI simply prompts for a different input or a quit signal. When the input is recognized but YANLI does not know a proper response, three default-response levels are provided. A default response can be given for each generic sentential input. This default is activated when an input falls into a category that does not have a proper response supplied for it. A default response of this type can be entered or changed any time that a response is being built by "(build-responses)." If a category has no default response specified for it, and input is given for which no response is available, then YANLI replies with a response that it finds in the file-system file "default-answer." This file can contain any textual response.

```
→(setq context 'sit-a)
→(build-responses)
..(help)


..((help 'a))
..q
→(setq context 'sit-b)

→(build-responses)
..(help)

..((help 'b))
..q
```

The context classification is set.
The response generator is activated.
The word "help" is the input sentence.
YANLI responds with a table in which the name is "sit-aiphelp_". (omitted)
The only response will be to activate help library "a". Quit to build next.
Set the situation in which help library "b" is appropriate.
Turn the response generator on.
Table is again presented. This time the name is "sit-biphelp_". (omitted)
Again, the only response is to activate the help. Quit.

Figure 8. Example of the Construction of a Context Sensitive Help Function.

Finally, if there is no default-answer file in the file system, YANLI replies, "I'm not programmed to respond to that."

Summary

YANLI is a small (about 4000 lines of code) versatile tool for building a customized natural language interface for programs in the Lisp environment. Although YANLI does not depend upon or use world knowledge, the ATNs that define its grammar can be modified to contain such knowledge. If they cannot be modified, new ATNs can be built using "make-atn." The addition of a response generator makes YANLI useful even in the absence of world knowledge. The response generator makes the parts of the input structure easy to access and makes possible the classification of sentential structures. Because of this ability, responses based upon the input can be stored in secondary memory for recall when needed. The responses can consist of text, executable expressions, or a combination of the two. The executable expressions can contain as data any information compiled by the response generator. These features make YANLI a useful natural language front end for a variety of applications, in particular those applications which have their own knowledge representation schemes.

Acknowledgments

I want to thank Lois Hawkes for her help in writing this paper and for her support and suggestions.

References

Christaller, T. 1982 An ATN Programming Environment In *Augmented Transition Networks*, ed L Bolc, 71–148 Berlin: Springer Verlag

Finin, T W 1977 The Planes Interpreter and Compiler for Augmented Transition Networks. In *Augmented Transition Networks*, ed L Bolc, 1–69 Berlin: Springer Verlag

Harris, M D. 1985. *Introduction to Natural Language Processing* Reston, Va : Reston Publishing.

Johnson, S C 1984. YACC: Yet Another Compiler-Compiler In *ULTRIX-32 Supplementary Documentation, Volume II*, 3–112 Nashua, N H.: Digital Equipment.

Woods, W A 1970. Transition Network Grammars for Natural Language Analysis. *Communications of Association of Computing Machinery* 13:591–606.