Natural-language interface for an instructable robot†

ROBERT ELTON MAAS AND PATRICK SUPPES
Institute for Mathematical Studies in the Social Sciences, Stanford University
(Received 8 December 1983 and in revised form 7 May 1984)

This article is concerned with the problems of understanding the grammar and semantics of English as it would be used to instruct a robot. We describe a working prototype system, programmed in Standard LISP, which translates English sentences into an operator language and then executes them. Imperatives become actions that are performed while declaratives become assertions that are checked for truth. We describe the context-free grammar and parser that are the first stage of translation, the postparser that uses context-sensitive information to eliminate the many bad parses permitted by the context-free grammar, and the table-driven final stage of translation. We also describe the semantics of the operators and arguments, including the context-referent mechanisms we invented to construct loops rather than the artificial means (such as labels and goto, or begin–end pairs) usually employed in programming languages. Finally, we discuss the general question of designing systems which build new procedures out of already-known procedures in ways that are similar to how a human being is taught new procedures in terms of old procedures.

The overall purpose of this article is to approach the problems of understanding the grammar and semantics of English by asking what is required to implement a program of instruction for robots. The use of robots rather than children, for example, is to keep important problems from being hidden by the verbal and perceptual facility of children. Children understand too much that we cannot make an explicit part of theory. Robots, at least the kind we consider as prototypes, are rather stupid and do not offer a helping hand to weak theories in the way that children do. We have also put the emphasis on executing procedures rather than the handling of declarative sentences in order to focus on bringing the analysis and understanding of language to the point of execution. We recognize, of course, that one can test the understanding of declarative sentences by asking for judgments of their truth value, but we are more concerned with procedures, partly because the kinds of responses required are more complex.

We are especially interested in the problem of instruction. A robot, like a child at any given stage, must have available a certain set of primitive procedures. A central problem of instruction is to teach the robot new complex procedures by formulating in English new procedures in terms of given ones. To certain readers this may sound like automatic programming. Later we shall discuss the ways in which it is different. In fact, the ideas for the instructable robot came out of the extended experience of one of the authors with the teaching of elementary mathematics.

Our work relates in different ways to various earlier efforts to develop language-understanding systems, to provide a procedural semantics for perceptual words and

† A longer version of this paper with extensive appendices may be obtained by writing to Professor Patrick Suppes, IMSSS, Ventura Hall, Stanford University, Stanford, California 94305, and requesting Technical Report No. 306.
phrases, or to write programs that learn from various forms of instruction. We say something about the first two topics here and delay further remarks about programs that learn until section 4.

Winograd's SHRDLU program (1972) is one of the best known natural-language understanding systems. Its aim is to be able to converse with a person about manipulating objects in a simple world of toy blocks. The system answers questions and accepts commands that are to be executed (in simulation). It is designed to take account of current context, both in the sense of recent past discourse and in the sense of the perceptual scene. Our system improves on SHRDLU most importantly in the interactive facilities for teaching our "robot" new complex procedures. In other respects our system is more restricted than SHRDLU because of our almost exclusive emphasis on interactive instruction.

The same general comparative remarks apply to such recent efforts as the Hearsay-II speech-understanding system (Earman, Hayes-Roth, Lesser & Reddy, 1980). This system has as a goal, the understanding of spoken language and thus embodies important features that we have not even considered. Apart from the handling of spoken language, the problem-solving capacities of Hearsay-II much exceed ours. Our efforts are somewhat closer to the HARPY system (Lowerre & Reddy, 1980) for understanding speech, in that both systems are more sharply focused than Hearsay-II. But even here the comparison is not a close one because HARPY has compiled knowledge as a central feature. The only significant virtue of our system in comparison with these two more powerful ones is the one already emphasized, i.e. the capability of interactive instruction to build new procedures.

The highly interactive features of our system are closer to central features of interactive theorem provers than to any of the three systems just discussed. Several detailed articles on such theorem provers are to be found in Suppes (1981a). Our interactive instruction, like interactive construction of a proof, can be viewed as a dialogue or conversation between program (or robot) and person, but unlike the three systems mentioned, general problems of knowledge representation are avoided. Complex proofs can be given or complex procedures can be constructed, but the framework for either is sharply delineated. What is critical is not knowledge representation but powerful methods of construction. Of course, neither the interactive theorem provers mentioned nor our system described in this article are nearly as powerful as we would like, but we do think that their current implemented interactive features compare favourably with the other systems discussed.

Another line of work pertinent to our efforts, even though it does not consist of developing a speech-understanding system, is the extensive work on procedural semantics of perception words, best exemplified in Miller & Johnson-Laird (1976). Their summary of control instructions of cognitive importance in human perception overlaps with our list of primitive atoms, although the earlier list in Suppes (1972) for our very restricted perceptual world is the actual initial source of our list. Miller & Johnson-Laird (1976) provide a great deal of invaluable analysis and comment pertinent to any language-understanding system that aims to handle any significant part of the perceptual language of English. Compared to the many issues they focus on, we have been able to consider only a small number.

This article naturally divides into four main parts. The first section deals with the grammar and parser. As we shall discuss later, the parser itself actually consists of a
NATURAL LANGUAGE INTERFACE

preparser, a main parser, and a postparser. The second section is concerned with semantics and translation. We translate parsed English into LISP operator language and then provide a semantic interpretation for the new primitive atoms we add to LISP. New primitive atoms are given a particular semantics relevant to the particular procedures being dealt with. Section 3 focuses on the execution of procedures and the interpreter. From the standpoint expressed above, we are not content with an abstract or even a partially abstract procedural semantics but consider it important to give a full analysis of the execution of procedures. Finally, section 4 deals with instruction from the standpoint of learning new procedures.

The procedures we deal with in this article are all ones that arise in teaching a child elementary arithmetic; we emphasize that we include a perceptual component. Before saying more, it will be useful to give a very simple example of the parsed English, the operator language translation, and the execution of the operator language translation of one of our simplest English commands.

English:
Look at the top number.

Parsed English:
(IMPATIVE VP+Advs look (PrepPh at (NP+Adjs number the top)))

Operator language translation:
(SEQUENCE (LOOK TOP) (CHECK NUMBER))

Execution:
Sequence...

(LOOK TOP)
I see "3" at row 1 (from the top) and column 1 (from the right).

(CHECK NUMBER)
Ok.

...end of sequence.

The overall dataflow for an interactive session with the robot is shown in Fig. 1. Prior to an interactive session, the whole training corpus can be compiled, which means applying the parser and translator to each different sentence in the corpus, to produce lookup tables relating input and output of parser and translator, and formatting the original corpus and the lookup tables for fast access later during the interactive session. (This compilation and formatting process is referred to as "batch mode" in the dataflow chart.) During the interactive session, sentences from the corpus can input from the database just mentioned, or new sentences can be typed interactively at the terminal or can be read from various program-debug rigs, as shown at the top of the chart. Each sentence is then parsed and translated, to yield the operator-language expression. The robot then recursively executes that expression in the environment of the example being worked out, asking the user for help in resolving ambiguities the robot could not resolve itself. Finally, the compiled and disambiguated program is given a name, written into a database of known algorithms, and may later be loaded back in to be applied to a different example without having to be compiled and disambiguated again.

The perceptual component of the procedures we consider is admittedly highly schematized. The perceptual plane is taken to be a coordinate grid of unit squares, and the recognition of a finite alphabet of symbols is assumed, with at most one symbol occurring in a give unit square at any one instant. On the other hand, the simplified perceptual scheme has enough elements of realism, especially in movement around
Fig. 1. Dataflow-interactive mode.
the grid in terms of current points of attention or gaze, that it has been used for the detailed study of eye movements in doing arithmetic exercises (Suppes, Cohen, Laddaga, Anliker & Floyd, 1982; Suppes, Cohen, Laddaga, Anliker & Floyd, 1983).

A training sequence of steps leading to the standard school algorithms for addition and subtraction formed the test bed of initial instruction for our robot program. The instructions were all in English and derive from those given in Suppes (1980). To illustrate ideas, let us restrict ourselves to the case of one-column addition. In principle, two registers suffice in our scheme of analysis. There is a stimulus-supported register that holds an encoded representation of a printed symbol to which the student—child or robot—is perceptually attending. In the present case the alphabet of symbols consists of the 10 digits and the underline symbol. As a new symbol is attended to by a change in the point of visual focus, a previously perceived symbol is lost from memory unless transferred to a second register which is non-stimulus-supported. A typical instruction for this transfer is “Remember this number”. In our formulation the student is assumed already able to add a one-digit number to a given number, but this capacity could itself be built up from more elementary operations such as that of counting.

1. Grammar and parsers

This section is divided into four parts, first the grammar, then the preparser, next the main parser, and finally the postparser.

1.1. THE GRAMMAR

There is nothing very notable about the grammar we have written. We had available on the Institute’s computer system a program written by Robert Smith some years ago, which will accept any context-free grammar and parse utterances appropriately prepared in terms of the given context-free grammar. The grammar we have written is strictly a parsing grammar. It would not be adequate for output of English, because we have not in the present formulation provided explicit indication of subject and verb agreement, verb tenses, and other central features. We do not pretend for a moment that this grammar is adequate to further developments but we have constructed it to have something in a framework that was already easily available to us. It needs to be refined considerably for future extensions but it is good enough to exhibit the main features of concern in this first article on our approach to procedural semantics. A typical top-level rule would be one for conjunction of sentences.

\[ S_1 \leftarrow S_0, \text{Conj} \ S_1 \]
\[ S_2 \leftarrow S_1, \text{Conj} \ S_2 \]

Another rule that generates a complex sentence from parts is the rule for \textit{then}:

\[ S \leftarrow S, \text{then} \ S. \]

We also have the following rules to distinguish between declaratives and imperatives. The rules for each of these are as follows:

\[ S_0 \leftarrow D \]
\[ S_0 \leftarrow I \]
\[ I \leftarrow \text{continue} \ S \text{ until } D \]
I ← continue S, until D
I ← VP
D ← NP VP
D ← Dadv NP.

Notice that the first two rules for imperatives introduce at the top level familiar, but rather special, English constructions of importance in our context. An example we used is “Continue doing everything from those steps up through the most recent step, until you see a space”.

Our fragment of grammar is rather standard, but we have also introduced a number of specific details to handle the kinds of procedures we have encountered in dealing with our particular subject matter.

12 PREPARSER

Robert Smith’s parser, called “CREATE”, requires that input be in a special form. To permit sentences to be typed in ordinary form, we wrote a program, which we refer to as the preparser, to convert sentences to the special form.

13 MAIN PARSER

The main parser program was written by Robert Smith, using SAIL (Reiser, 1976), a version of Algo 60. Various modified versions of it exist, CREATE being the one we use for the robot. It has no detailed documentation, but a general description is given in Smith, Rawson & Smith (1974). It uses a reasonably efficient algorithm, the Cocke-Younger one (Younger, 1967) for compiling a context-free grammar into a set of tables that permit rapid determination of all parses of a given sentence, avoiding the laborious backtracking that would be needed by a brute-force algorithm that used the grammar directly without compiling it.

When the program is started, it first loads the grammar in from a file (the syntax of this grammar file is given later) and compiles it, checking for loops. A loop is where some symbol (token) is defined to be exactly itself directly or indirectly. For example, “S₁ ← S₁ P” is legal because each time that production is used an extra P must occur, but “S₁ ← S₁” is not legal. Similarly, “S₁ ← S₂” and “S₂ ← P S₁” is a legal combination, but “S₁ ← S₁” and “S₂ ← S₁” is not a legal combination.

The grammar file consists of one production per line, each production being in three parts: (1) the symbol being decomposed, (2) the decomposition, i.e. syntax, (3) the resultant output, i.e. semantics. The general form of a production is (symbol) ← (syntax)(tab) “(semantics)”.

Here is a sample line from our current grammar:

VP ← VP Conj VP ”(; 2; ; 1; ; 3;)”.

This defines one form of a Verb Phrase to be a smaller Verb Phrase followed by a Conjunction followed by another smaller Verb Phrase. For example, if “look” is one Verb Phrase, “see” is another, and “and” is a conjunction, then “look and see” is parsed as a larger Verb Phrase. Output consists of a three-element list consisting of first item #2 (the Conj) then item #1 (the first VP) and finally item #3 (the final VP). Note that the three items in the list are the results of recursively applying the semantics. For example, if the semantical representation of “look” is “LOOK”, that of “and” is
"SEQUENCE", and that of "see" is "SEE", then that of "look and see" is "(SEQUENCE LOOK SEE)".

It is possible to give the program a command that will enable a trace of the syntax tree for each parse found, as is familiar in context-free grammars. Unfortunately the program has no similar feature for showing the semantics (output).

1.4. POSTPARSER

The output from CREATE is not suitable for our purposes. First of all, CREATE often generates multiple parses, due to deficiencies in our grammar. These are placed one after another, and only look-ahead can detect when we have reached the end of the multiple parses and started to read the next sentence in the file. Also, the output is not prettyprinted, so it is hard for us to see from the output it makes which parses (if any) are good when debugging the grammar. Finally, there are usually many obviously bad parses in addition to the one correct parse and a few almost-correct parses. With our present context-free parser and our present grammar that does not know about case, mode, and many other things, we needed to flush the obviously bad parses before further processing the parsed sentences. We thus decided to write a postparser, which collects all multiple parses into a data structure, rates them according to acceptability of the parse, removes those parses which are inferior to others, generates a warning when no parse achieves a desired acceptability level, and prettyprints the single accepted parse or the list of parses not inferior to any other parse.

The general mechanism of rejecting inferior parses is by weighted demerits. Each proposed parse is recursively examined, looking for any pattern that indicates an obviously bad parse, and checking the type (case or mode) of an overall expression against the types of its parts. Each bad pattern or each type mismatch causes a demerit to be assigned to the parse as a whole. Each type of demerit has a number associated with it, and these numbers for demerits are added to decide how inferior the parse is to a "perfect" parse. That is, a weighted sum of demerits is used to rate the badness of each parse. Among the various parses competing for the title of "best" parse for a given sentence, any parse inferior to another is eliminated, leaving one or more parses, all with the least weighted demerit.

At present, all demerits have a weight of one (except for one class that has zero weight), and every sentence in our original corpus parses with a demerit score of zero. When new sentences are added to implement new robot algorithms, sometimes there is no parse at all that has zero total demerits. In that case, a parse with the smallest weight of demerits is chosen. Definitions and examples of each class of a demerit are given later.

After eliminating all inferior parses for a given sentence, there is usually exactly one parse remaining, which is then flattened and prettyprinted to the output file. This flattening involves bringing obviously parallel structure to the same level of expression (parenthesis), such as all adjectives modifying a noun, all adverbs modifying a verb or other word, and subject-verb-object of a transitive sentence. These structures were not already flattened when they left the main parser because the grammar was written to minimize the number of rules by decomposing structure one step at a time instead of all at once. For example, we did not parse "now look two spots down" by a special rule that allowed exactly one adverb phrase preceding a verb and two adverb phrases following it ("two spots" is a single adverb phrase here, and "down" is the other
adverb), and “look two spots down” by a completely different rule that allowed zero
adverb phrases preceding and two adverb phrases following a verb, and “look down”
by yet a third unrelated rule. Instead we built all parses of this type by two rules, one
that allowed a single adverb phrase preceding a verb and gave back a verb, and one
that allowed a single adverb phrase following a verb and gave back a verb. These two
rules suffice for an arbitrary number of adverb phrases preceding or following a verb,
but instead of getting a flattened parse like “(V + Advs look now (two spots) down)”
we get a tree-like structure such as “((VP + Adv (VP + Adv (VP + Adv (VP + Adv look now) (two
spots)) down))”. If the main parser had a way to make dotted pairs—technically the
basic elements in a list in LISP—which could be strung together to make a list of
variable length, it would not be necessary for the main parser to produce such
inscrutable output. But dotted pairs are not supported in the main parser. The only
way to get rid of the mess of nested 3-element lists and produce a single list of variable
length is by some additional processing after the main parser. The postparser does this.

Here is an example of a parse before and after flattening by the postparser. We have
manually reformatted (prettyprinted) the S-expressions for maximum legibility. (A
fully-parenthesized LISP expression is called an S-expression.)

Original English text: Continue looking at the next spot down adding and remembering
until a bar appears.

Output of the correct parse from the parser (ignoring about fifteen other parses that
also appear before demeriting). Note that “A, B and C” parses as “(SEQUENCE A
(and B C))” instead of the more reasonable but not possible in the main parser
“(SEQUENCE A B C)”:

(IMPÉRATIF (CONTINUE-UNTIL
(SEQUENCE (IMPÉRATIF
(VPR + Adv (VPR + Loc looking
(PrePhe at (NP + Adjs spot the next)) down))
(and (IMPÉRATIF (Shortform adding))
((IMPÉRATIF (Shortform remembering)))))
(Subj + Verb (NP + Adjs bar a) appears)).

Correct parse after flattening, which includes changing the idiom “A B and C” to have
the desired parse, and bringing sub-expressions up to the same level as single-argument
parent expressions, as well as flattening multiple adverbs:

(IMPÉRATIF CONTINUE-UNTIL
(SEQUENCE (IMPÉRATIF VP + Advs looking
(PrePhe at (NP + Adjs spot the next)) down)
(IMPÉRATIF Shortform adding)
((IMPÉRATIF Shortform remembering)))
(Subj + Verb (NP + Adjs bar a) appears)).

Here are the two classes of demerits most frequently used by the postparser, with an
example of each. The number after each class name is the count of times this demerit
is applied in our whole corpus of training steps. These are in descending order, so the
most important (troublesome) cases are first.
MIXED-MODE (292)
(IMPERATIVE CONTINUE-UNTIL
(SEQUENCE (IMPERATIVE VP+ Advs looking down)
(or (IMPERATIVE VP+ Advs looking (PrepPh for (NP+ Adjs number a)))
(DECLARE Subj+Verb
(NP+ Adjs bar a)
(and (Shortform adding)
(Shortform remembering)))))
(Subject+Verb you (TV+ NP see (NP+ Adjs bar a))).

In the above example, an "or" joins an imperative and a declarative. This is clearly wrong because conjunctions are supposed to join phrases that are parallel in structure, such as two imperatives, or two declaratives. There are other obvious errors in the above parse, such as a declarative sentence (inside the "or" part) whose verb consists of present participles instead of present-tense forms, and the use of "or" instead of "and" to terminate a sequence, but our postparser does not presently detect these errors.

CONTINUE-IN-DOUNTIL (77)
(DO-UNTIL
(TV+S continue
(SEQUENCE
(IMPERATIVE VP+Advs looking (PrepPh at (NP+Adjs spot the next)) down)
(IMPERATIVE Shortform adding)
(IMPERATIVE Shortform remembering))
(DECLARE Subj+Verb (NP+Adjs bar a) appears)).

The problem here is that the grammar is redundant. Any sentence that parses as (DO-UNTIL (TV+S continue... ...)) will also parse another better way in which "continue" and "until" are treated as a single idiom. The robot is set up to understand this idiom rather than to attempt to handle "continue" inside an "until" structure. The grammar-production for "until" by itself is set up to handle sentences that do not contain the word "continue" such as "Look down until you see a number", but a context-free grammar can not easily exclude the use of one production inside another.

For practical work it is almost certain that certain exception lists will need to be added, because it is not possible to make a "perfect" postparser that eliminates 100% of the bad parses for correct reasons. We have done just that, creating a file of bad parses that our program cannot presently eliminate by any normal means. In some cases the bad parse is obviously wrong, sometimes in rather subtle ways, but there is no simple mechanical rule for eliminating them. In a few other cases there is a simple rule but modifying the daisy-chained postparser code to try to include these cases was deemed too difficult to be worthwhile doing at this time, considering that we plan to rewrite the postparser to be table-driven anyway. These latter will be included in a proper way when the postparser is rewritten to be table-driven. Both these non-rule cases and these rule-but-not-programmed cases are included in an explicit exception list. When the parsed English exactly matches any of these exception-list parses, it is assigned the demerit KNOWN-REJECT-PAR. We present here one example for illustration.
English: "Write out the answer in the next spot open."
The bad parse is:

(IMPERATIVE TV+NP
 (TV-IDIOM write out)
 (NP+Adjs answer the
  (PrepPh in (NP+Adjs spot the next open)))).

This parse implies that something is to be written out, but without a location where
to write it, and that the existing thing is the answer in the next open spot, i.e. the
answer is already in the next spot open before writing it out to location unknown. The
correct parse for this sentence is:

(IMPERATIVE VP+Adv
 (TV+NP (TV-IDIOM write out) (NP+Adjs answer the))
 (PrepPh in (NP+Adjs spot the next open)))).

This good parse provides a place for the answer to be written, and the object to be
written out is merely the answer, which does not have to be already in the next spot open.

2. Semantics and translation

To get to semantics we translate from parsed English to operator language. The
semantics is then easily specified in terms of the standard syntax of LISP—we used
Utah Standard LISP (Hearn, 1969; Frick, 1978). We first say something about semantics
and then turn to translation.

2.1. SEMANTICS OF ATOMS

In our applied version of LISP, three kinds of primitive atoms occur. First, there are
operators. Examples would be LOOK-AT and ADD. Second, there are actors. In this
case, an example is DEFAULT-NUMBER. Third, there are data. An example would
be BOTTOM. Use is made of the standard atom NIL but this never appears in any
printout and is in one sense not a proper part of our operator language.

The semantics of operators follows one of three patterns. One pattern is a standard
recursive evaluation of EXPRs in LISP, whereby each of the arguments to a function
is recursively evaluated before calling the function, for example, the standard LISP
function LESSP for numerical less than. The second is treating the arguments of the
operator as data and passing them to special coercion routines before performing the
main operation. An example is the operator SUBTRACT in our robot operator
language, which must have its arguments coerced to numbers by calling COERCE-TO-
NUMBER on each. Another example is the operator LOOK-DIRDIS which passes
its first argument to COERCE-DIRECTION-TO-DEL-COLOR and its second argu-
ment to COERCE-DISTANCE-TO-NUMBER. The third pattern is a case analysis on
the operator’s arguments and in this instance nothing else is called to analyse the
semantics. For example, the operator LOOK-AT checks the form of argument before-
hand and then passes on a sub-expression to another function.

The semantics of actors mainly consists of search and return of a datum. For example,
the actor DEFAULT-SUM searches the memory stack as a database and returns the
datum found as is.
2.2. MODEL THEORY

From a model-theoretic standpoint the semantics is based upon the grid world of our arithmetic examples, which contains a perceptual component based upon a small fragment of mental functions as, for example, the operator *remember* and a very elementary fragment of arithmetic used in teaching the algorithms of addition and subtraction, for example, add 2 to 7. In addition, there is a recursive evaluation similar to that of arithmetic of Boolean expressions which constitutes the logical component of the model. Finally, there is a control aspect. Do something whether a given expression is true or false. The truth or falsity of the expression changes the execution behavior.

In this discussion we have distinguished perceptual, mental, arithmetical, and logical terms, but in the semantics of the operator language no explicit distinction occurs.

2.3. TRANSLATION

The translation table consists of a set of token names, and for each token name a sequential list of rules. Each rule consists of a template (also called the "syntax") and a result (also called the "semantics"). One of the token names is TOP which corresponds to the top-level parsed sentence that comes out of the parser. Other token names correspond to sub-expressions, a matter discussed below.

The list of rules associated with TOP is applied at the top level of the parsed sentence that is fed into the translator. Sometimes the TOP rules may also be used for some sub-expression that is really a sub-sentence embedded in the overall sentence, for example, if the top-level sentence is a conditional (if . . . then . . .) or a loop (do . . . until . . .). Other token names correspond to sub-expressions that need different rules, for example, prepositional phrases and nouns.

To translate a sentence, the overall expression is matched against the list of rules associated with TOP, except only the templates are used, not the semantics. The templates of the rules are searched sequentially until one is found that matches. Typically some idiomatic form of English sentence structure is matched *en masse*, sort of a gestalt recognizer for idioms. The first template that matches determines the rule to be used. All later rules in the list, even if their templates also match, are ignored.

Each template has constant-atomic parts, constant-structure parts, and variable parts. The constant parts are the places where an exact match must occur to accept the template match. In the case of an atom, exactly that atom must appear. In the case of a dotted pair (the basic unit for building lists), a dotted pair must appear, and then the two branches of the template are matched against the corresponding branches of the actual expression. All constant parts must match exactly for the template to be accepted. That is, if at any point in the matching process a constant-atom in the template corresponds to a dotted pair or to the wrong atom in the expression, or if a dotted pair in the template corresponds to an atom in the expression, the match of the expression against that template is immediately deemed a failure, and the matching attempt moves on to the template of the next rule in the list. The only place where the corresponding sub-expressions in the template and the expression may differ is where a variable occurs in the template. Here anything may appear in the expression, atomic or dotted-pair.

Note that backtracking is implicit in the template-matching process. However, once a template has been selected, no further backtracking at that level occurs.
Once the correct template is selected, the variable parts are used. Each variable part of the template has a different name that corresponds to one of the token names in the overall translation table. The same name also performs as a pseudo-variable, which is now bound to the sub-expression at the corresponding place in the overall expression of parsed-English. (In LISP this is called “destructuring”, as in the DESEQ and LET macros.) Here is an example:

Parsed English: (IMPERATIVE VP+Advs look (PrePPh at (NP+Adjs number the top)))
Template: (IMPERATIVE VP+Advs look (PrePPh at [NOUN]))
Successful match—resultant binding: [NOUN] = (NP+Adjs number the top)

The constant parts of the template are “IMPERATIVE”, “VP+Advs” et al., in addition to the dotted pairs that are used in the internal representation. The one variable part is [NOUN]. The template matches the parsed English because “IMPERATIVE” et al. all match the corresponding places in the parsed English. The spot marked by [NOUN] is ignored during matching. Then, after getting the successful match, the pseudo-variable “[NOUN]” corresponds to the parsed-English sub-expression “(NP+Adjs number the top)”, so that binding is made.

After the top level of template-matching and pseudo-variable-binding is done, the translator is called recursively on each of the bindings. Instead of “TOP”, the name of the variable is used, for example, “NOUN” in the above example. The same process is repeated as many levels deep as there are further pseudo-variable bindings. The results (described below) are passed up, and the pseudo-variable is rebound to the final result from translating “(NP+Adjs number the top)” according to the rules associated with “NOUN”. In this example, the result will be:

New binding: [NOUN] = (INTERSECT-CUES (OBJECT NUMBER) (LOCATION TOP))

Now the original rule is consulted again, except instead of the template, the result (semantics) part is used. With the above example, in our current translation grammar, the semantics happens to be:

Semantics = (LOOK-AT [NOUN])

The “new binding” of [NOUN] given just above is now substituted into the semantics, replacing the occurrence of [NOUN] given in the semantics. The result is:

(LOOK-AT (INTERSECT-CUES (OBJECT NUMBER) (LOCATION TOP)))

When a template has no pseudo-variable spots, then no binding occurs, no recursive call to the translator occurs, and no substitution in the semantics occurs. Instead, the semantics is returned as is, as the result of the parse. This is how the recursion “bottoms out”, avoiding infinite recursion depth. In the above example, the [NOUN] happened to invoke such a variable-less rule, thus the recursion depth was two, one level for the TOP-level matching, and one level for the NOUN-level matching.

When a template has more than one pseudo-variable spot, each of them is in turn recursively translated, each with the set of rules associated with the name of the pseudo-variable. When all are done, each of the results is substituted for the correspondingly-named pseudo-variable in the semantics, and that result is returned.
It is possible to enable a trace feature whereby all of this recursive translation is printed on the terminal, with each level of recursion causing a level of indentation. (To make the trace easier to read, the details of each recursion are indented a half-level. Currently a level of indentation is four spaces and a half-level is two spaces.) First, the token name (TOP at the outermost level of recursion) and the expression are printed. Then the matching syntax is found and printed. Then variables are bound and the binding list is printed as an ASSOC list, i.e. each element is one binding, with the CDR of that element being the variable name and the CDR being the sub-expression to which it is bound. Then the recursion occurs on each binding, shown at deeper level(s) of indentation. Then the new bindings which came back from the recursion are printed as "Results" in the same way the initial bindings are printed as "bindings". Then the result (semantics) part of the rule is shown, and finally the result of substituting the "Results" into the "Semantics" is shown preceded by "\(=\)". Figure 2 shows one of these traces, for the simple example shown above.

```lisp
TOP = (IMPERATIVE VP+Advs look (PrepPh at (NP+Adj+ number the top)))
Syntax = (IMPERATIVE VP+Advs look (PrepPh at [NOUN]))
Bindings = ((NOUN NP+Adj+ number the top))
NOUN = (NP+Adj+ number the top)
Syntax = (NP+Adj+ number the top)
Semantics = (INTERSECT-CUES (OBJECT NUMBER) (LOCATION TOP))
== (INTERSECT-CUES (OBJECT NUMBER) (LOCATION TOP))
Results = ((NOUN INTERSECT-CUES (OBJECT NUMBER) (LOCATION TOP)))
Semantics = (LOOK-AT [NOUN])
== (LOOK-AT (INTERSECT-CUES (OBJECT NUMBER) (LOCATION TOP)))
```

**FIG. 2.** Simple example of translation.

Figure 3 shows a more complicated example where more than one pseudo-variable appears at one level of recursion. (Note that some lines are too long to be printed in one piece. They are broken at the right margin and continue on the next line, slightly indented.)

There is one special case in the semantics. If an exclamation mark appears where an atom (word) would ordinarily appear, the rest of that sub-expression is treated specially: call some function to perform additional processing before returning the translation. The next element in the list, at the same level as the exlamation mark, is the name of the function to be called. Ordinarily this is T2-REC which invokes yet another recursive call to the translator besides the normal recursive binding and substitution. The remaining elements are arguments to that function, which first get all variables substituted just like the ordinary semantics part of the rule. Thus, any such extra call occurs after the normal recursion which gives sub-results that are the new bindings for the variables, and just after the argument sub-expression result has been formed by substituting the sub-results for the variables in the argument sub-expression. But this extra call occurs before passing this sub-expression up to the next level of building the overall expression to return.

When T2-REC is the function being called, there are exactly two arguments, the first being the token name associated with the desired rules to use, and the second being the form to be retranslated in that way (according to those rules). In the above example, at the spot where we omitted the semantics because they used a mechanism that had not yet been explained, that is what happened. Here is that portion of the
English.
Add this number to the number you remembered.

Parsed-English.

\[(\text{IMPERATIVE } \text{VP+Adv}s (\text{TV+NP add (NP+Adjs number this)})
\text{ (PrepPh to}
\text{ (N+roleuse (NP+Adjs number the) (that you remembered THAT\&SLOT))))\]

Translator running with trace on:

\(\text{TOP} = (\text{IMPERATIVE } \text{VP+Adv}s (\text{TV+NP add (NP+Adjs number this)}) \text{ (PrepPh to}
\text{ (N+roleuse (NP+Adjs number the) (that you remembered THAT\&SLOT)})\])

\(\text{Syntax} = (\text{IMPERATIVE } \text{VP+Adv}s (\text{TV+NP add [NOUN1]})) \text{ (PrepPh to [NOUN2])}\)

\(\text{Bindings} = (\text{[NOUN1] NP+Adjs number this})
\text{ (NOUN2 N+roleuse (NP+Adjs number the) (that you remembered}
\text{ THAT\&SLOT)})\)

\(\text{NOUN1} = (\text{NP+Adjs number this})
\text{ Syntax} = \text{[NOUN]}
\text{ Bindings} = (\text{[NOUN]} \text{ NP+Adjs number this)})
\text{ NOUN} = (\text{NP+Adjs number this})
\text{ Syntax} = (\text{NP+Adjs number this})
\text{ Semantics} = \text{DEFAULT-NUMBER}
\text{ == DEFAULT-NUMBER}
\text{ Results} = (\text{[NOUN] DEFAULT-NUMBER})
\text{ Semantics} = \text{[NOUN]}
\text{ == DEFAULT-NUMBER}

\(\text{NOUN2} = (\text{N+roleuse (NP+Adjs number the) (that you remembered THAT\&SLOT)})\)

\(\text{Syntax} = \text{[NOUN]}
\text{ Bindings} = (\text{[NOUN]} \text{ N+roleuse (NP+Adjs number the) (that you remembered}
\text{ THAT\&SLOT)})\)

\(\text{NOUN} = (\text{N+roleuse (NP+Adjs number the) (that you remembered}
\text{ THAT\&SLOT)})\)

\(\text{Syntax} = (\text{N+roleuse [NOUN] (that MOVER) [VERB] THAT\&SLOT})\)

\(\text{Bindings} = (\text{[NOUN]} \text{ NP+Adjs number the})
\text{ MOVER} = \text{you}
\text{ Syntax} = \text{you}
\text{ Semantics} = \text{ROBOT}
\text{ == ROBOT}
\text{ VERB} = \text{remembered}
\text{ Syntax} = \text{remembered}
\text{ Semantics} = \text{REMEMBER}
\text{ == REMEMBER}

\text{Results} = (\text{[NOUN] DEFAULT-NUMBER})
\text{ MOVER} = \text{ROBOT}
\text{ VERB} = \text{REMEMBER}

\text{[Semantics omitted because it uses a feature not yet explained.]} == \text{SOMETHING-ACTED-UPON ROBOT REMEMBER ANY-NUMBER)

\text{Results} = (\text{[NOUN] SOMETHING-ACTED-UPON ROBOT REMEMBER ANY-NUMBER})
\text{ Semantics} = \text{[NOUN]}
\text{ == SOMETHING-ACTED-UPON ROBOT REMEMBER ANY-NUMBER})

\text{Results} = (\text{[NOUN1] DEFAULT-NUMBER})
\text{ NOUN2 SOMETHING-ACTED-UPON ROBOT REMEMBER ANY-NUMBER})
\text{ Semantics} = (\text{ADD [NOUN1] [NOUN2]})
\text{ == ADD DEFAULT-NUMBER (SOMETHING-ACTED-UPON ROBOT REMEMBER ANY-NUMBER})

Fig. 3. Complicated example of translation.
above trace now:

Results = ((NOUN . DEFAULT-NUMBER)  
(MOVER . ROBOT)  
(VERB . REMEMBER))

Semantics = ((SQMETHING-ACTED-UPON [MOVER] [VERB]  
( ! T2-REC ANYIFY [NOUN])))

ANYIFY = DEFAULT-NUMBER
Syntax = DEFAULT-NUMBER
Semantics = ANY-NUMBER
= = ANY-NUMBER
= = (SOMETHING-ACTED-UPON ROBOT REMEMBER ANY-NUMBER)

As shown by the above trace, the results from recursion were:

NOUN = DEFAULT-NUMBER
MOVER = ROBOT
VERB = REMEMBER

The semantics called for building the trivial expressions ANYIFY which is a constant and [NOUN] which becomes DEFAULT-NUMBER, then passing these two arguments to T2-REC. The call then occurs, which has the effect of translating DEFAULT-NUMBER according to the ANYIFY rules. The result is ANY-NUMBER. This expression is then passed up to the main expression being built. This is a list of four elements, the first being the constant SOMETHING-ACTED-UPON, the second and third values of MOVER (i.e. ROBOT) and VERB (i.e. REMEMBER), and the fourth this expression passed up (i.e. ANY-NUMBER).

Here is another example:

Semantics = ( ( ! T2-REC NOUN-TO-PREDICATE [NOUN]) DEFAULT-OBJECT)

Suppose that “NOUN” was originally bound to some parsed-English sub-expression that returned “BAR”. When running the semantics, first T2-REC is called with “NOUN-TO-PREDICATE” as first argument and “BAR” as second argument. The effect is as if the pseudo-variable “NOUN-TO-PREDICATE” existed and was bound to the sub-expression “BAR”, except that here “BAR” comes not from the parsed-English but from the translation of something according to the “NOUN” rules. Anyway, the translator is called recursively, using the rules for “NOUN-TO-PREDICATE”, with “BAR” as input, and suppose the result is “BARP”. Now “BARP” is substituted into the expression in place of the ( ! T2-REC NOUN-TO-PREDICATE [NOUN])”, and the result is:

Result: (BARP DEFAULT-OBJECT)

This result is then returned up to the next level.

3. Interpretation and execution

Because the operator language is standard LISP with the exception of the primitive atoms, interpretation has strict LISP sense in our system. The CAR of an expression
says what to do. On the other hand, evaluation is not standard LISP. In LISP terminology everything we do is assumed to be an FEXPR (an F expression). Evaluation does not take place without looking at the CAR of an expression. The CAR of an expression is ordinarily a function, and this function decides when and how to interpret the arguments and thus when and how to do evaluation. For example, in standard LISP consider the expression

(PLUS X 3).

The standard LISP method is to recursively evaluate the arguments first. For example, suppose the variable "X" had been assigned earlier the value 8. Then the function would be evaluated to give the sum 11. In contrast, when PLUS is treated as an FEXPR the trace of a computation would look the same but the function PLUS would be called first.

It is also true that, in general, execution is that of standard LISP but this does not mean too much because each primitive FEXPR has its own execution. Of course, this is irrelevant for actors and operators which have no arguments. It is also true that statistically most operators with arguments act like EXPRs which have the standard LISP recursive evaluation of arguments.

3.1. COERCING DATA

Sometimes the argument given to a function does not meet the needs of that function. For example, an arithmetic function, needing numbers as arguments, might be given characters or variables as arguments. In a strongly-typed language such as PASCAL it is hard to get around this problem, but in an object-oriented language such as LISP where every object carries its type around with it at runtime and where functions exist for checking the type of an object at runtime, it is not too hard to coerce data of the wrong type into data of the right type at runtime.

When the robot is requested to do an arithmetic function such as addition, COERCETO-NUMBER is called on its arguments to convert them to numbers if they are not already. COERCETO-NUMBER calls the function MAY-COEERCETO-NUMBER to do most of its work. MAY-COEERCETO-NUMBER either converts the object to a number or returns a failure indication. If it fails, COERCETO-NUMBER then generates an error message, sets a flag indicating a number was needed (to be used by an error-recovery procedure not yet written), and aborts the current step.

If the object is already a number, then of course it is kept as is and the coercion succeeds. The most common case where the object needs coercion to a number is when it is a character of text from the input data matrix. For example, the character-object !5 must be converted into the number 5. The only other case currently handled is when the text of the command contains a number-word such as "two" or "ten". This is converted into the appropriate LISP numerical representation.

The opposite sort of coercion happens when the command is to output something, for example, writing a character in the arithmetic-example matrix. In this case only characters can be written, so whatever the argument is must be converted to a character. The way it does this is to do an EXPLODEDEC and check to see if the length is 1. If not, that object needs more than one character to print, and cannot be coerced into a single character. If the length is indeed 1, then that single element, the single character in the print-name, is returned as the result of COERCETO-CHARACTER. If the
datum was already a character, taking the only element of the EXPLODEC gets back to that character again.

When we want to move the point of attention somewhere, via the LOOK command, or when we want to perform an input or output at some location relative to our current location, various words having to do with direction must at some point be converted into actual coordinates. That is the job of COERCE-LOCAT-TO-COORDS. First it converts keywords into coordinate offsets. According to the semantics of our operator language, HERE is (0 0), LEFTWARD is (1 0), DOWNWARD is (0 1), UPWARD is (0 -1), etc. Then it adds these offsets to the current coordinates to get the new coordinates. These are later used as the actual arguments to the operator function such as LOOK.

3.2. LOCATING REFERENCES TO PAST STEPS

A major facility of our English-understanding robotics approach is the ability to create program loops without the artificial methods of standard programming languages. For example, BASIC has every line numbered for editing, and GOTOs can specify one of these line numbers. FORTRAN has lines numbered only when they will be referenced, and they do not have to be in numerical order. ALGOL and LISP have alphanumerical labels, again only when needed. But all of these methods are artificial compared to the way students in school are taught algorithms for arithmetic. Even explicit loop primitives used in structured languages such as PASCAL are unnatural. Imagine the very first time you are guided through column addition you are told “O.K., now here’s a loop that will be executed over and over until an exit condition is satisfied, and here are the steps in the loop . . . ” before any of the individual steps has been performed even once.

We adopted the following approach, which is the closest we have come to the natural method. First we take the “student” through the individual steps in the loop one at a time, with no reference to any of the steps being “in a loop” and no mention of specially marking any of the steps with a “label”. We require only that the student keep a list of the steps in sequence, possibly translated from the description we give to his own internal notation. Then at the point when we have guided the student through all the steps of the loop once and we are ready to collect them into a “loop”, we refer to the first step in the loop, not by numerical order but by context. We can either tell the student all in one command where the loop starts and ends (it always ends with the most recent step before the loop-making command) and what the exit condition is, or we can separate the label-making (context-matching) part from the loop-making part, as we found necessary due to deficiencies in the parser. But in both cases we need a way to specify some past step based on context (syntactic) information. This is the job of NEWMAT, the new pattern-matcher for past steps. (Previous versions did not work in difficult cases, so the routine had to be completely rewritten using a different approach.)

Before discussing the internal workings of NEWMAT, let us discuss its external interface and how it is used. NEWMAT take as argument a list of cues into past steps. These may be complete repeats of the steps, paraphrases, or very truncated skeletons of them. For example, if the original step was “look down at the next number” the cue given to NEWMAT may be “look down at the next number” or “looking down at the next number” or “looking for the next number” or “looking down”, etc. In all
cases NEWMAT will find (or at least is supposed to find) the correct step being referred to. Given a list of cues into past steps, it returns a list of numbers of those steps as currently listed in the memory of the robot-student's memory. Note that because earlier some steps may have been collapsed into single steps, these numbers do not necessarily correspond to the sequence of commands originally given to the robot, and later, if further collapsing is done, these numbers will become invalid. They must be used very soon, typically in the current command or the next, to be valid.

When everything is combined in a single command, it is in the form "Continue ... until ..." which invokes the function O-CONTINUE-UNTIL. NEWMAT is called, and the numbers that come back from it are immediately used to fetch the corresponding steps from the memory. These steps are formed into a loop which is executed in single-step mode with exit check at the start and end and between steps as described below. Upon completion of the loop, any just-preceding steps which match final steps in the loop (after the exit condition) are rolled into the loop, causing those steps to disappear from the top level of memory and to appear at the start instead of end of the loop.

When the reference and the loop are separated, the reference is invoked by "Refer back to when you ..." while the loop itself is invoked by "Continue doing everything from that step up to the most recent step, until ...". The "Refer back ..." calls NEWMAT which returns a list of step numbers as above. But here all the numbers except the oldest are thrown away, and the oldest is set as the value of a global variable, which represents the label that would be present in an ALGOL program. The "Refer back ..." step is not saved in memory for later reference because it is not something that might later be repeated. This is the one exception (currently) to the rule that the student (robot) remembers all past steps executed. Then the "Continue doing everything ..." resolves the argument "that step" by fetching that saved number (the index, i.e. label, of the referred-to step), and resolves the argument "the most recent step" by fetching the number of the most recent step. It then creates a list of all numbers between those two bounds, i.e. a list of indexes of all steps to be included in the loop. Finally it does the same thing as O-CONTINUE-UNTIL, fetching all steps whose indexes are in the list, making a loop, stepping thru it, and rolling preceding steps into it where possible.

Thus NEWMAT converts cues into the indices of the actual steps to be brought into the loop, and those indices are used to fetch the past steps to make the loop. Either just the steps mentioned or all steps, from the oldest mentioned to the most recent executed, are included, depending on whether the "continue ... until ..." or the separate "refer back ..." and "continue doing everything ..." syntax was used.

Now we will describe the internal workings of the current versions of NEWMAT. It is given two items of information, its argument consisting of cues to past steps, and the global variable listing past steps in reverse chronological order. The first thing it does logically is to convert each list into a list of lists of keywords. That is, each cue and each past step is converted into a list of keywords, so that each list of cues or list of steps is converted into a list of lists of keywords. In the case of the list of cues this is done en masse, via MAPCAR, but in the case of the past steps this is done in a lazy way, using a "lazy evaluator" to do only as much of MAPCAR as is needed in actual processing. We shall not go into the details of the lazy MAPCAR, except to say that it creates a partial list of results with a blip at the end indicating the rest has not yet
been computed and pointing at the place in the original list where not-yet-processed data reside. When CDR is attempted at the point where the blip stands, one more element is fetched from the original list, processed into keywords, and patched into the result list, then the blip is advanced past that new processed data and the pointer to the original data is advanced past the data which have now been processed. The reason for this lazy MAPCAR instead of simply calling MAPCAR to convert all past steps *en masse* is that often a loop refers only to two or three past steps, and if there are thirty steps in all it is a gross waste of computing to process all thirty steps if the pattern-matching algorithm is not even going to look at more than two or three steps.

After the list of past steps (via lazy MAPCAR) and the list of cues (via normal MAPCAR) have been converted to lists of keywords, the cues are reversed so they are in reverse chronological order (like the past steps). Then these lists of lists of keywords are collated to find the most recent place in reverse past steps where something like the cues occurs. The first (latest) cue is matched against the first (latest) past step. If the match succeeds, then the next (earlier) cue is matched against both the same and the next (earlier) past step. If either succeeds, recursion continues until a mismatch occurs or the list of cues is exhausted. If failure occurs at any level, a re-try of any alternative pending at a higher level is tried. If a failure at the top level occurs (either directly or by all lower-level alternatives failing somehow), the pointer into the past steps is advanced and the whole process is retried. The effect of all this is that if a success occurs, it corresponds to the MOST RECENT sequence that matches, and because it immediately returns at the top level, all older sequences that might also match are suppressed. Thus NEWMAT finds the most recent CONTIGUOUS sequence of past steps that matches the cues given.

For example, suppose after conversion to keyword-lists, the past steps (in reverse chronological order) are:

-1 (SPACE WRITE ZERO)
-2 (LOOK LEFT)
-3 (LOOK DOWN)
-4 (WRITE NUMBER)
-5 (LOOK RIGHT)
-6 (LOOK RIGHT)
-7 (REMEMBER NUMBER)
-8 (SPACE WRITE ZERO)
-9 (LOOK LEFT)
-10 (WRITE NUMBER)
-11 (LOOK RIGHT)

...(steps farther back irrelevant here)

Suppose the English command that specified a reference point in these past steps was something like “Refer back to when you looked to the right, wrote out a number, looked to the left, looked for a space, and wrote a zero”, which converted to keywords and reversed becomes:

(((WRITE ZERO) (LOOK SPACE) (LOOK LEFT) (WRITE NUMBER (LOOK RIGHT)))


The most recent place where (WRITE ZERO) matches is at −1. NEWMAT tries to find (LOOK SPACE) at the same or preceding step, i.e. at −1 or −2, and finds it at −1 only (it is not an exact match, but let us assume it is above the threshold). It then tries to find (LOOK LEFT) at −1 or −2, and finds it only at −2. It then tries to find (WRITE-NUMBER) at −2 or −3, but can not find it at either, causing this whole attempt to fail. The next most recent place where (WRITE ZERO) matches is at −4 (again an inexact match). But (LOOK SPACE) does not match at −4 or −5 so this aborts quickly. Next farther back (WRITE ZERO) matches at −8, (LOOK SPACE) matches at −8 or −9. In the former case, (LOOK LEFT) matches at −9, (WRITE NUMBER) at −10, and (LOOK RIGHT) at −11, completing the desired match. Possible matches starting with (LOOK SPACE) at −9 are never tried now. The result returned is the list of indices (−8 −8 −9 −10 −11), of which only the −11 is used in the case of "Refer back to..." as here.

Only one detail of NEWMAT remains to be defined: in the above recursive algorithm, when considering whether the keyword list from a particular cue matches the keyword list from a particular past step, what is the criterion for a successful match? First of all, keywords are coerced into a standard form of paraphrasing. Thus "looking", "looked", and "look" all come out the same. This makes it possible to find a match when the original step was "Look at the next spot down", while the cue came from "Refer back to when you looked at the next spot down" or "Continue looking down, adding and remembering". All versions of the pattern matcher do this coercion the same. The original pattern matcher scored a success whenever any keyword in the cue, after this coercion, matched any keyword in the step, after the same coercion. This worked fine for the main loop of addition and also for the little loops in both addition and subtraction, but was too lenient (willing to find a match) for the main loop of subtraction, finding a false match consisting of only the most recent six steps when actually the past 30 steps were supposed to be included. We fixed it by requiring a particular percentage of keywords in the cue to match any keywords in the past step. Note that we could not do the converse, requiring a particular percentage of keywords in the past step to match keywords anywhere in the cue, because of the possibility that the past step is a loop composed of several small steps and that the cue is referring to one of those small steps. We needed a successful match even in cases when one of the steps referred to in the cue was merely one very small part of the large loop that was a past step.

We tried various thresholds, various required percentages of keywords to match, to see which values of the threshold got a false match that masked the deeper correct match, which values skipped over the false matches to reach the correct match, and which values skipped past even the correct match to find no match at all. We found that the false match for subtraction matched half the keywords, thus we needed a threshold greater than 0.5 to get past it. The correct match matched all the keywords; thus, any threshold greater than 0.5 and less than 1.0 would be satisfactory. But then we went back and checked the threshold for the other loops, and found that the correct match for a small loop found only half the keywords in one of the cues, thus requiring a threshold less than 0.5. This put us in a dilemma. The threshold had to be less than 0.5 for one match to work and greater than 0.5 for another match to work. In both cases the English was quite reasonable, so we decided not to force the English to be modified to make the pattern matching come out consistently. Our only easy alternative
was to "tune" the program, having a different threshold in the two cases. This can be a dangerous thing to do, because the program is so carefully tuned on the test data it has been developed with that it cannot handle other test data that are identical as far as human judgement is concerned but for which the tuning is not right. We have tried as much as possible to make our mechanisms general, to match parts of sentences in reasonable ways rather than simply to hardwire our program to understand exactly the complete sentences we used as our training data.

Our solution was interaction with the user. The program has a default threshold that works in most cases, although not, for example, in the main loop of the subtraction algorithm. It uses that threshold to find the match, and tells the user both the complete step that it found and the number of steps from that step to the most recent step. This gives the user enough information to decide whether that is the right step or not. If not, the user must supply a new threshold to try. Again the program gives feedback and the user considers the result. When the threshold results in finding the correct step and the user confirms it, the program proceeds with the rest of the processing as before. This method is not totally satisfactory since it requires the user to understand the concept of threshold, an internal aspect of the robot's program, in addition to understanding the problem domain. Perhaps we should change this so the user merely says whether the step found is correct, too far back or too recent. The robot would then use that information to pick a new threshold silently instead of asking the user (teacher) to give it a new threshold.

Finally, we describe the workings of the single-step mode for loops mentioned above. Initially, the exit condition is tentatively located at all possible points in the program, at the start and end and also between each two adjacent steps. Then when executing the loop, if one of these copies of the exit condition is satisfied, the user is asked if it is the correct one. If it is not, that copy is deleted so that the user will not be bothered with the exact same query during a later pass through the loop. If it is the correct one, all others are deleted, leaving just that one exit condition at the correct location in the loop. If there is only one remaining copy of the exit condition when the user answers "no" to the query, the remaining loop contains no exit condition at all and thus can never terminate. The single-step mode detects this error and aborts immediately, flagging the whole loop as invalid. When copies of the exit condition are not satisfied, however, they are left as is, since they may be satisfied during later passes through the loop.

4. Learning and instruction

A notable feature of current work in computer science and artificial intelligence is the relative neglect of a sustained effort to develop concepts and techniques of learning, an exception being Michalski, Carbonell & Mitchell (1983). A liberal definition of "learning" includes any change in behaviour. Under this definition a computer "learns" when it is given a new program to execute. In fact our work instructing a robot is very similar to programming in that we give the robot a sequence of commands which the robot compiles into a more machine-oriented language and then executes or stores for future execution. Our use of a reduced set of English is not even unique. Recall that COBOL was designed to be like English, and several companies now advertise programming languages that are claimed to be fragments of English. But there are important differences between our efforts to create an "English language programming language"
and these others. Whereas prior attempts have tried to make the English precise in the same way that programming languages make their source language precise, we allow ambiguities to exist in our English source language. During debugging, rather than requiring the user to go back and modify the English source to resolve ambiguities, we resolve ambiguities during a test run. Most of the resolution is obtained not by direct interactions with the user but by interactions with the test data. Thus if the program has more than one interpretation, but only one interpretation makes sense in conjunction with the test data, the robot can resolve the ambiguity without needing any help from the user-instructor. Even where not all but one interpretation can be eliminated, the number of possible interpretations can be greatly reduced, giving the user a small list of interpretations to choose from. Furthermore, the possible interpretations are presented to the user as practical questions such as ‘is this the right time and place to exit the loop’ in the course of working through the test data, instead of as \textit{in vacuo} questions about the structure of the program. (An example is discussed later.) Thus the user is relieved of the burden of mentally simulating the program to figure out where the exit condition needs to be, the robot does the test run for him and asks for help at exactly the point in the test run when it is easy for the user to see what needs doing and answer the query correctly. Also the robot is relieved of the burden of anticipating all possible problems before starting execution. Instead the robot can plod ahead as if the program were unambiguous, and ask for help only when it is already bumping into possible trouble. We believe this is analogous to a teacher giving instruction to young children, where a child is incapable of anticipating all the trouble he will have in learning a new algorithm, and the teacher is incapable of knowing ahead of time all the possible ambiguities in the student’s mind and giving unambiguous instruction from the start. Ambiguity of instruction is perhaps the most important feature differentiating teaching children from programming computers. The child resolves residual ambiguity by asking questions, and so should the robot.

It is not our objective in this article to give an overall view of types of learning or a classification of possible tasks. Without attempting to give a less liberal definition of learning than the one mentioned above, we restrict ourselves to the learning that is closely associated with instruction. We have in mind tasks of the kind that correspond to much instruction that is given in school, especially to young children; for example, algorithms of elementary arithmetic and approaches to beginning reading. Many other perceptual and motor-skill tasks fall under this category of being explicitly instructable. We emphasize, of course, that much learning does not fall under this concept of instruction. It does represent, in our judgment, an area of great importance and we are concerned here only with it. The central feature of such learning from instruction is, we believe, the synthesis of complex procedures from simpler ones in accordance with the instructions given, in English or some other natural language.

We now give a couple of concrete examples of new procedures that we synthesized from primitive procedures available. (We emphasize that a limitation of the present system is that new procedures must be built up directly from primitive procedures rather than calling other complex procedures that have been previously built up. The main reason is that we do not yet have a mechanism for the procedures we build to take arguments. They are effectively inline-code macros rather than parameterized subroutines.) From a formal standpoint these examples are trivial. They are meant to
NATURAL LANGUAGE INTERFACE

illustrate three things: the use of English for formulating the new procedures, the
difficulties we encountered, and a comparison of the English formulation of the new
procedure to the LISP S-expression generated as the final product of the compilation
of the English. Note the compilation involves three parts, (1) the translation of the
English into operator language which initially contains some ambiguities, (2) the
interaction of that translation with the problem domain, which resolves some of the
ambiguities, and (3) interaction with the user (playing the role of the teacher) which
resolves the remaining ambiguities. This is different from traditional programming of
computers where the source language must have no ambiguities.

The first example takes any addition problem in standard column format and simply
places in the answer the first row of digits—for simplicity we assume that the first row
has as many digits as any other row in the exercise. The English that successfully
constructed the algorithm was the following:

Look at the top number.
Remember this number.
Look one space down until you see a bar.
Look one space down.
Write the number you remembered.
Look at the top of the next column to the left.
Refer back to when you remembered the number and looked down.
Continue doing everything from that step up thru the most recent step, until you
see a space.

The S-expression which is the final product of compilation is as follows:

\[
((\text{LOOK-AT} \ (\text{INTERSECT-CUES} \ (\text{OBJECT} \ \text{NUMBER}) \ (\text{LOCATION} \ \text{TOP})))
(\text{LOOP} \ (\text{REMEMBER} \ \text{DEFAULT-NUMBER})
 (\text{LOOP} \ (\text{LOOK-DIRDIS} \ \text{DOWNWARD} \ (\text{DISTANCE} \ 1))
  \ (\text{IF} \ (\text{BARP} \ \text{DEFAULT-OBJECT}) \ (\text{DONE})))
 (\text{LOOK-DIRDIS} \ \text{DOWNWARD} \ (\text{DISTANCE} \ 1))
 (\text{WRITE-OUT} \ (\text{SOMETHING-ACTED-UPON} \ \text{ROBOT} \ \text{REMEMBER}
  \ \text{ANY-NUMBER}))
 (\text{LOOK} \ (\text{INTERSECT-LOCATION-CUES} \ \text{TOP} \ \text{LEFTWARD}))
 (\text{IF} \ (\text{SPACEP} \ \text{DEFAULT-OBJECT}) \ (\text{DONE}))))
\]

It is easy to establish the correspondence between the lines of the S-expression and
the English. The most important general feature is the use of the “continue” idiom to
establish the loop.

The English is not as precise as the S-expression but it is more readable and much
easier for the user to understand. The directions given to the robot here are more
explicit than one would use with a child but not enormously so. It would be easy to
device an experiment to give something very similar to six- or seven-year-old children.

Because of the length of the record of the interaction by which we constructed this
simple algorithm successfully, we have not presented it. However, there are one or
two useful comments about the kinds of difficulties we encountered with the current
version of our program in constructing the algorithm given above. First, we had some
difficulty with the program’s accepting the English we first used for the third line above.
Our first try was "Look downward until we get a bar". Unfortunately the three words "downward", "we", and "get" were not in the grammar. The next attempt was "Look down one space until you see a bar". We got several parses of this instruction but no operator-language translation. The difficulty was that the grammar treated "down" occurring "before one space" as a preposition. In the parse of the sentence that was translated into the proper operator language and given above, "down" occurs after "one space" and is classified as an adverb.

Permuting columns. This example is more complicated but still quite simple. Given an addition exercise in standard format with two columns but with not all numbers required to be two-digit numbers, the algorithm permutes the columns. It is understood that a zero is first to be placed in the tens column of a one-digit number.

The 23 English commands that led to successful construction of this algorithm are the following:

Look at the top number.
Look one space to the left.
Look one space to the left.
Look one space to the right.
If you see a space write out a zero.
Look here again.
Remember this number.
Look one space to the left.
Write out the number you remembered.
Look one space to the right.
Look one space to the right.
Remember this number.
Look one space to the left.
Write out the number you remembered.
Look one space to the left.
Remember this number.
Write out a space.
Look one space to the right.
Look one space to the right.
Write out the number that you remembered.
Look one space down.
Look one space to the left.
Refer back to when you wrote a zero.
Continue doing everything from that step up thru the most recent step, until you see a bar.

We feel that the readability of the procedure described in English is very much better than that of the generated S-expression:

```
((LOOK-AT (INTERSECT-CUES (OBJECT NUMBER)(LOCATION TOP)))
 (LOOK-DIRDIS LEFTWARD (DISTANCE 1))
 (LOOK-DIRDIS LEFTWARD (DISTANCE 1))
 (LOOK-DIRDIS RIGHTWARD (DISTANCE 1))
 (LOOP (IF (SPACEP DEFAULT-OBJECT)(WRITE-OUT 0))
     (LOOK DEFAULT-LOCATION)(REMEMBER DEFAULT-NUMBER))
```
It is evident from this example that there are many obvious improvements to be made in our system to provide a flexible environment for interactive instruction. Some are at the level of language. For example, it is desirable to extend looking to any finite number of spaces, so that lines (2) and (3) could be replaced with the single command "Look two spaces to the left". A second feature concerns access to memory. Our "robot" can access only the top of the stack and the topmost stack item satisfying some property such as being a number or a sum or a difference, etc. Also the stack cannot be purged of items that are no longer needed. This was no problem in handling the algorithms of addition and subtraction, but is already a problem in the present example. There are various grammatical devices in ordinary English for making multiple references to memory—anaphoric reference being among the most prominent. We see no problem in principle of matching the internal workings of our robot program to these various devices, as long as the relevant grammatical theory has been developed.

We promised earlier to give an example of using test data to disambiguate loop control, and we now do so using the above program as an example. That program contains a major loop with 18 steps plus one exit test which happens at the end. But the English description of the algorithm does not tell that it is bottom-tested. Indeed top-tested loops are more commonly useful, and tests in the middle are useful too in some cases. In fact, there are 19 possible places to put the exit condition, and the English does not give any information as to which of them is correct. With full mathematical analysis it is possible to eliminate seven of them, leaving 12, but this is very difficult analysis which requires full knowledge of the semantics, in particular that having written a number at one spot and moving two units to the right and two to the left we are back at the same spot which is still a number and thus cannot be a bar now. Such mathematical analysis is pushing the state of the art of program verification. Also, mathematical analysis does not give us any idea which of the 12 is more likely to be the desired exit location. Our program, by comparison, requires no mathematical analysis, yet still reduces the search space to 12, and in this example
gives the correct answer as the second choice. When processing these particular test
data (or any similar example with bars only at the bottom), the first time a bar is seen
is just after the next to last step, and the second time is after the final step of the same
iteration. Therefore the user has very little trouble picking the correct location to insert
the exit test in the final algorithm.

As we indicated already in the introduction it is not our present intention to extend
our system so as to make it capable of general knowledge representation. But as in
the case of interactive theorem provers, at this stage of development our aim is to
handle efficiently and easily the rather specialized facts and procedures of a given
restricted domain. Every major component of the system reported on in this article
can obviously be improved, and we hope to do so in the future.

References

speech-understanding system: Integrating knowledge to resolve uncertainty ACM Compu-
sing Surveys, 12, 213–253.
Computation Group, Memo TR-2.
Stanford, Calif.: Stanford University.
Harvard University Press.
Stanford, Calif.: Stanford University.
of meaning. Proceedings of the 1974 Conference of the Association for Computational Linguis-
tics. Amherst, Mass.
and Philosophy. Proceedings of the 4th international Wittgenstein Symposium, Kirchberg
am Wechsel, Austria 1979, Vienna: Hölder-Pichler-Tempsky.
Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social
Sciences.
movements in arithmetic performance. In GRONER, R. & FRAISSE, P., Eds, Cognition and
YOUNGER, D. H. (1967). Recognition and parsing of context-free languages in time \( n^3 \).
Information and Control, 10(2), 189–208.