MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 277

February 1973

A LINGUISTICS ORIENTED PROGRAMMING LANGUAGE

Vaughan R. Pratt

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-70-A-0362-0003.

ABSTRACT

A programming language for natural language processing programs is described. Examples of the output of programs written using it are given. The reasons for various design decisions are discussed. An actual session with the system is presented, in which a small fragment of an English-to-French translator is developed. Some of the limitations of the system are discussed, along with plans for further development.

A Linguistics Oriented Programming Language

Vaughan R Pratt

1. Overview

This paper presents some aspects of work done at odd intervals over the past two years, first at Stanford and then at MIT, on a project to develop a programming language suitable for writing natural language processing programs. The relevant acronym is LINGOL, for Linguistics Oriented Language. Similar projects such as COMIT (Yngve 1963), and its successors METEOR (Bobrow 1964) and SNOBOL (Farber 1964) no longer reflect the state of the art of computational linguistics; indeed, they do not rise above the remark that computational linguistics is concerned with processing text strings. The issue addressed in these pages is that of the programming technology appropriate to the syntax-semantics interface, an artifice that arises in the phrase-structure paradigm for natural languages. A secondary issue, to be dealt with elsewhere, concerns the relative merits of various parsing strategies for phrase-structure oriented grammars, and the development of a parsing algorithm superior to both the Earley and Cocke-Kasami-Younger procedures. (See Aho and Ullman (1972), p. 314).

Following Winograd's (1971) lead, we begin by giving some examples of the output of programs written in LINGOL. The point of having a programming language is to make programming less painful for all concerned. The interesting property of these programs is that two of them were written in quite a short space of time by students with no experience in either LINGOL or linguistics. Another program (the French translator) was designed, written and debugged from scratch for demonstration purposes by the author from 3:00 a.m. to 8:00 a.m. of the morning he was scheduled to give a talk on it.

The first program was written in September 1970, to test out the first version of LINGOL. It was a sort of "deep structure" analyzer which attempted to make syntactic remarks about sentences (Figure 1). The grammar used in it served as the basis for the next two programs.

The system languished for six months until a graduate student, Bill Faught, took advantage of it for a project in an A.I. workshop. He took two weeks to write an English-to-German translator (Figure 2).

Later, Faught decided to do some serious work on question-answering systems, and soon produced a comprehension program (Figure 3) that relied on a relational model of the world in which all related concepts were represented in a graph as vertices linked by two-way, labelled edges. Recently he has produced considerably more impressive results, but it is more appropriate that Faught himself report on them.

The French translator (Figure4) was written by the author early in 1972, for demonstration purposes. The program consisted of a page of grammar and semantics, a page of

```
NONE OF MY FRIENDS WERE EATEN BY A LION .
ASSERTION:
        FALSE
                ACTOR: LION
                ACT: EAT
                OBJECT: FRIENDS
                        PORTION: SOME
                        OWNER: I

THE AUTHORS OF NONE OF THESE BOOKS ARE FRIENDS OF PETER .
ASSERTION:
        FALSE
                SUBJECT: AUTHORS
                        SPECIFY: THE
                        AUTHORS OF: BOOKS
                                PORTION: SOME
                                PLACE: HERE
                BE: FRIENDS
                        FRIENDS OF: PETER

IF A MAN CAN EAT A DOG A HORSE CAN EAT A SMALL CAT .
ASSERTION:

                ACTOR: HORSE
                ACT: EAT
                OBJECT: CAT
                        SMALLNESS: SMALL
                IMMEDIACY: POTENTIAL
                CONDITION:

                                ACTOR: MAN
                                ACT: EAT
                                OBJECT: DOG
                                IMMEDIACY: POTENTIAL

HOW QUICKLY CAN YOU SWIM .
QUESTION:
        SPEED:
                ACTOR: YOU
                ACT: SWIM
                IMMEDIACY: POTENTIAL

EAT A HORSE .
COMMAND:

                ACTOR: YOU
                ACT: EAT
                OBJECT: HORSE

DO YOU LIKE ICECREAM .
QUESTION:

                ACTOR: YOU
                ACT: LIKE
                OBJECT: ICECREAM
```

Figure 1. "Deep Structure" Analyzer.

```
SHE SLEEPS WITH HIM BECAUSE SHE LOVES HIS CAT.

SIE SCHLAEFT MIT IHM WEIL SIE SEINE KATZE LIEBT


HE IS IN THE HOUSE.

ER IST IN DEM HAUS


HE HITS THE BALL TO THE DOG.

ER SCHLAEGT DAS BALL ZU DEM HUND


THE CAT IS IN THE TREE BY THE HOUSE.

DIE KATZE IST IN DEM BAUM NEBEN DEM HAUS


THE STUDENT WILL SLEEP BETTER WHEN HE UNDERSTANDS THE IDEA.

DER STUDENT WIRD BESSER SCHLAFEN WENN ER DIE IDEE VERSTEHT


THE LITTLE OLD MAN LOVES THE RED HOUSE BECAUSE HE CAN SEE THE HORSES.

DER KLEINE ALTE MANN LIEBT DAS ROTE HAUS WEIL ER DIE PFERDE
SEHEN KANN


WHILE THE FLOWER IS OLD THE APPLE IS YOUNG.

WAHREND DIE BLUME ALT IST IST DER APFEL JUNG


SHE WILL GIVE HIM A FLOWER IF HE EATS THE APPLE TODAY.

SIE WIRD IHM EINE BLUME GEBEN WENN ER DEN APFEL HEUTE ISST


THE HORSE THAT EATS BAD APPLES IS A SICK HORSE.

DAS PFERD DAS SCHLECHTE AEPFEL ISST IST EIN KRANKES PFERD


IF A HORSE CAN EAT A DOG A MAN CAN EAT A SMALL CAT.

WENN EIN PFERD EINEN HUND ESSEN KANN KANN EIN MANN EINE
KLEINE KATZE ESSEN
```

Figure 2. The German Translator of W. Faught.

```
        JACK  BUILT  A RED HOUSE. HE LOVED RATS. MORTIMER WAS JACK 'S
RAT.  HE LIVED IN THE HOUSE THAT JACK BUILT.
        JILL  BUILT  A BLUE HOUSE. SHE LOVED CATS. SPOT WAS A CAT WHO
ATE MORTIMER. JACK HATED JILL BECAUSE SPOT ATE MORTIMER.
        FIDO  WAS A BROWN DOG. HE CHASED SPOT. SPIKE WAS A BLACK DOG.
HE CHASED JILL.

WHAT IS THE COLOR OF THE DOG THAT CHASED THE CAT.
BROWN

DID JACK LOVE MORTIMER.
YES

WHO ATE MORTIMER.
SPOT

WHAT IS THE COLOR OF THE HOUSE IN WHICH THE RAT THAT THE CAT ATE LIVED.
RED

WHY DID JACK HATE JILL.
BECAUSE SPOT EAT MORTIMER

WHO DID THE BLACK DOG CHASE.
JILL

DID JILL LOVE ANY ANIMALS.
YES

 MEN HATE RATS.  DID JILL DISLIKE MORTIMER.
YES
```

Figure 3. Faught's Question-Answering System.

```
I LOVE MY WIFE.
J' AIME MA FEMME

SHE LOVES ME.
ELLE M' AIME

WE HAVE SOME BIG RED DOGS.
NOUS AVONS QUELQUES GRANDS CHIENS ROUGES

WE LOVE OUR DOGS.
NOUS AIMONS NOS CHIENS

OUR DOGS LOVE US.
NOS CHIENS NOUS AIMENT

BIG DOGS LOVE BEAUTIFUL RED TABLES.
LES GRANDS CHIENS AIMENT LES BELLES TABLES ROUGES

SMALL DOGS LIKE PRETTY LITTLE TABLES.
LES PETITS CHIENS AIMENT LES JOLIES PETITES TABLES
```

Figure 4. The 5-Hour French Translator.

dictionary and a page of useful functions for conjugating verbs, arranging agreements of articles and adjectives, performing liason and so on, so it was not particularly large. The point of it was

(i) it was easy to write;

(ii) it was sufficiently succinct to be suitable for exhibition; and

(iii) it dealt competently with that part of English for which it was defined.

It is easy to claim that, since this is a toy translator, it says nothing about the real world. This is certainly true with respect to polysemy. However, it is false with respect to extensibility of grammatical rules; we shall later demonstrate the striking effects obtained on adding very simple rules. More recently, another four hours of work gave a self-tutoring capacity to the program (Figure 5). Notice how unknown words are correctly classified as to part of speech before the program requests information.

We have been basking in these examples somewhat vicariously. It is very much like explaining the advantages and disadvantages of FORTRAN by exhibiting the output of some FORTRAN programs. Thus the reader should only infer from these examples the existence of LINGOL, and a lower bound on what can be achieved with it; he should infer its quality or lack of it not from here but from the following.

```
THE POLICEMAN GIVES THEM A TICKET.

WHAT IS "THE (OR A) POLICEMAN"?
LE GENDARME
WHAT IS "TO GIVE"?
DONNER
WHAT IS "THE (OR A) TICKET"?
LE BILLET
LE GENDARME LEUR DONNE UN BILLET.
1945. MILLISECONDS.

THE LITTLE GIRLS WANT A RED PIG.

WHAT IS "THE (OR A) GIRL"?
LA FILLE
WHAT IS "TO WANT"?
VOULOIR
CONJUGATE IT
VEUX VEUX VEUT VOULONS VOULEZ VEULENT
WHAT IS "THE (OR A) PIG"?
LE COCHON
LES PETITES FILLES VEULENT UN COCHON ROUGE.
1561. MILLISECONDS.

PIGS IS PIGS.

LES COCHONS SONT DES COCHONS.
773. MILLISECONDS.

I HAVE THE PEN OF MY AUNT.

WHAT IS "THE (OR A) PEN"?
LA PLUME
WHAT IS "THE (OR A) AUNT"?
TANTE
WHAT IS ITS GENDER (MASC OR FEM)?
FEM
J' AI LA PLUME DE MA TANTE.
1157. MILLISECONDS.

I WANT THE BUREAU OF MY ONCQUIT
◆
(HELLO)
TYPE SENTENCE FOLLOWED BY . ! OR ?

I WANT THE BUREAU OF MY UNCLE.

WHAT IS "THE (OR A) BUREAU"?
LE BUREAU
WHAT IS "THE (OR A) UNCLE"?
L'ONCLE
WHAT IS ITS GENDER (MASC OR FEM)?
MASC
JE VEUX LE BUREAU DE MON ONCLE.
1225. MILLISECONDS.
```

Figure 5. The 9-Hour French Translator, in Ignorance mode.

2. Terminology and Perspective

Let us set the stage preparatory to giving some definitions. We need a paradigm for computational linguistics programs, and we choose the <u>translation</u> paradigm as best describing the LINGOL system. The translation paradigm characterizes natural language processing programs as translators from the natural source language to some natural or formal target language, whether French, LISP, structural descriptions, predicate calculus, conceptual dependency diagrams or what have you. No loss of generality is entailed here, for by simply making the target language a programming language, any other paradigm may be conveniently emulated. The obvious competitor is the <u>stimulus-response</u> paradigm, in which the input is seen as a stimulus that elicits an action. Again no loss of generality can occur, since a possible action is to emit an utterance. The main advocate of this paradigm is Narasimhan (1969), although it appears to be the implicit paradigm in many extant programs. We prefer the former paradigm for no very good reason, although we do find it easier conceptually to manipulate and characterize utterances rather than actions. In particular, in the programming methodology to be described, large items are gradually built up from smaller ones, and it is tricky to cast this in a stimulus-response format.

Within the translation paradigm we shall identify two main phases, cognitive and generative. The cognitive phase is parsing, in which the input is preprocessed until it is in a form convenient for operation on by the generative phase, which then produces the translation as output. The paradigm itself does not require that one phase run to completion before the other can start. Indeed, Winograd's (1971) program makes effective use of feedback from the partial results of his generative routines in guiding the cognitive routines, by attempting to build a semantic structure for, say, a Noun Group, before continuing with the parsing.

We are now prepared for the definitions. By syntax is meant all aspects of the source language involved in the cognitive phase, including such things as phrase structure rules and semantic markers. By semantics we refer to what is involved in going from the source language (after the syntactic preprocessing) to the target language. By pragmatics we mean knowledge about the universe of discourse, and the local context, that may be consulted by both the cognitive and generative phases as they make decisions.

Each of these three concepts has been used many times in the literature, with varying shades of meaning and precision, so we are not redefining previously well-defined terms. Rather, we see three main aspects to the programs written in LINGOL, and found three reasonably uncommitted terms with which to label them. (The first two definitions coincide more or less with those of Winograd (1971), so we are not too far afield.)

(It may seem paradoxical to include semantic markers in syntax, but this is just the consequence of our usage of the word semantics as opposed to that of , say, Katz and Fodor (1964). With respect to our usage, semantic markers represent an attempt to encode a tiny fragment of pragmatics into syntax (or into linguistics, to use the Katz and Fodor terminology, and their equation SEMANTICS = LINGUISTICS - SYNTAX). We do not want to make value judgments about such an encoding; the example simply serves to illustrate the perspective induced by our definition.)

3. Design Philosophy

There is not one philosophy in LINGOL, but three, each tuned to the requirements of the three concepts defined above. In the current version of LINGOL, the philosophies are roughly as follows.

3.1. Syntax

Although this paper is concerned mainly with the semantic component of LINGOL, it behoves us to consider syntax since the cognitive phase's output is the generative phase's input. The central decision to be made here is the choice of representation for this output. It seems to be necessary to discover the relations between the words of the sentence, or the phrases of the sentence, or the entities denoted by those words or phrases. Corresponding to each of these possibilities are dependency structures (Hays 1964, Simmons 1964), phrase structures (almost everybody) and conceptual dependency networks (Schank 1970). Actually the first two are not mutually exclusive, since it is perfectly reasonable to construct structures which contain all the information of both techniques. We shall use the term <u>syntactic structure</u> to refer to such a coalition, to distinguish it from a <u>concept structure</u>.

LINGOL is meant to be a practical system suitable for export and immediate use by practising computational linguists. The technology for phrase structure is far advanced over any other technology, and every successful program for the past eight years or so has unashamedly used it. Also, it is fairly easy to convert a phrase structure system to a syntactic structure system, by tagging each phrase with the corresponding governing word together with pointers to the dependent phrases (and hence words).

For these reasons, the decision was made to use phrase structure as the output of the cognitive phase, leaving the other representations as projects to be experimented with in the future. It is worth noting at this point that the idea of a concept structure is a very powerful one, especially in combination with Fillmore's (1968) notion of case, as suggested by Shank (1970). The notion of phrase concatenation is nowhere near as rich as that of case-based relations between concepts. On the other hand, this does not make phrase-structure a hopeless loser; in principle it is possible to construct these relations during the generative phase. However, Shank's point is that the information so discovered is vital to the cognitive phase. More recent phrase-structure systems, including those of Bobrow and Frazer (1969), Woods (1969), Winograd (1971) and the system described here make provision for discovering this sort of information while building the phrase structure. This immediately raises the question, why not build the concept structure anyway, since this information is being discovered? This point seems unanswerable, and is an excellent area for more research. In the case of LINGOL, we have a half-answer, in that we have developed what we feel is very nice programming methodology for dealing with phrase structures during the generative phrase. An avenue for research is to see if this methodology carries over to concept structures.

Given that LINGOL is based on phrase structure, the next issue is that of the user's language for describing how that phrase-structure is to be built. The two criteria here are expressive power and ease of use. For our first iteration of LINGOL, since we were more interested in rapidly developing the semantics technology, we opted to sacrifice expressive power for ease of use if necessary. This corresponds in a way to Woods (1968) and Charniak (1972) assuming the existence of some sort of parser and continuing from there. The differences are firstly that both addressed pragmatic issues while we address semantic, and secondly that whereas they made up their own parsed output, LINGOL is equipped with a parser, on the philosophy that it is easier to type unparsed than parsed sentences, and that no harm is done when the parser gangs [illegible] agley, which in practice occurs satisfactorily infrequently anyway.

The user's language for the cognitive component was therefore chosen to be context-free rules, since these are very easy to write. They have exactly the same expressive capacity as Woods' (1969) transition networks. Moreover, just as Woods extended the capacity of these networks by allowing the user to specify operations on registers, so do we permit the user to supply code to give hints to the parser whenever it is about to apply a rule. This code has access to the part of the tree built so far by the parser and relevant to the rule in question, and also to the user's data base, or pragmatics (which seems to make semantic markers unnecessary as a special feature of LINGOL). The form of the hint is a grunt of approval or disapproval, at a volume appropriate for the particular hint, and in this repect is just like Winograd's (1971) numerical treatment of ambiguity. So far, however, none of the programs written in LINGOL have made more than trivial use of this feature, in sharp contrast to the use made of the features in the semantics stage.

With respect to the actual parser used, the syntax philosophy is that the parser should be transparent to the user, to within the representation of the parts of the tree to which the user's code has access during the cognitive phase. This philosophy has enabled us to run without alteration each of a number of different LINGOL programs in conjunction with various parsing algorithms. The details of these parsers and experiments are beyond the scope of this paper.

3.2. Semantics

In programming his semantics, the user should be able to work without the distracting detail of parsing, tree representation, and ambiguity. The point of identifying the cognitive and generative phases is to isolate these issues logically in order to achieve this division of labor. Whether writing an English-toFrench translation program or a question-answering system, there are many details to worry about that have absolutely no relevance to the cognitive phase; the myriad idiosyncrasies of French grammar and style, the various searching algorithms and inference rules that are tightly coupled in a QA system to the surface structure information, and so on. Without some method in this large-scale madness, progress is bound to be slow.

Furthermore, we believe that a high level of performance will be forthcoming from the cognitive phase of, say, machine translation programs, long before a similarly impressive level is attained by the generative phase. This is partly because comparatively little work is being done on generative aspects of MT, but more because it is inherently harder to say something with good grammar and style than it is simply to understand what is being said (at least explicitly!). The cognitive phase can ignore most details of style, and many details of grammar. In every program written so far with LINGOL, the generative component has been about three times the size of the cognitive component, and our prediction is that this ratio will increase as each phase is improved.

In taking this point of view, we are following a different philosophy from that of Winograd (1971), who makes use of strong interaction between the syntax and semantics components, which is one of the more notable features of his program. However, the result has been to produce a program whose details are lost in the richness of this interaction, and I have heard Winograd mutter when looking at a part of the program for "BE", "I don't remember writing that".

For the moment we are willing to sacrifice whatever additional power this approach has to offer for the sake of being able to write clean, modular, transparent semantic code. However, we do not believe that in order to restore this power we need to restore this interaction. Instead, we plan to rely eventually on strong interaction between syntax and pragmatics, leaving semantics as the cognition-independent arena. This is not just passing the buck; since we see semantics as being more complex than syntax, we are trying to divide the work-load more evenly to keep all modules reasonable small. How syntax is to consult pragmatics is material for future research. Our point is that the bulk of semantics is irrelevant to syntax.

The issue now is simply, how does one write programs that operate on trees (the output of LINGOL's cognitive phase)? This issue has been addressed by computer scientists in connection with compiling for the past ten years, and the discipline of syntax directed translation has gradually emerged. An early syntax directed translator is that of Warshall and Shapiro (1964). They used the <u>tree-walk</u> paradigm, in which the semantics consists of programs that tell a pointer to move up, down or across the tree and occasionally output information. Floyd (conversation) has commented that the technique was much too clumsy for practical applications when compared with techniques that tied the semantics to the syntax rather than to the output of the syntax. It is alarming to find Winograd using this approach in his program, which we conjecture would be made more transparent by adopting a more rule-oriented and less tree-oriented approach.

Some theoretical work has been done on syntax-directed translation, notably by Lewis and Stearns (1968), Knuth (1968), and Aho and Ullman (1972). Knuth's paper is of interest in that it deals with the problem of passing information up and down a tree, using the notions of inherited (from above) and synthesized (from below) attributes. All of these studies suffer, from the computational linguist's point of view, in that they deal with the microcosm of computer source and target languages, in which the former can be made a compromise between the user's needs and the syntax-directed technology, and the latter is a relatively well-defined, reference-poor language when compared with, say, French.

Knuth's inherited and synthesized attributes come closest to meeting our needs. The problem with these attributes lies with his mechanism for moving them around a tree. Every node through which information is passed must make explicit provision for forwarding it, even if it is irrelevant to that node.

For example, consider:

No mother of such twins has time to relax.

The mother of no such twins has time to relax.

The mother of such twins does not have time to relax.

The mother of such twins has no time to relax.

(The second sentence is inspired by a study of negation by Klima (1964). It should be said in a tone of horror, with the emphasis on "no", before it sounds correct.)

In each case, what is being negated is the whole sentence, yet the negation marker can be almost anywhere in the sentence. This implies that a large number of rules will have to make provision for passing up a negation marker.

This problem can be circumvented by using global variables instead of Knuth's attributes. Now all that is needed is for the negation marker to set a negation variable, and for the semantics at the syntactic clause level to read it.

However, consider the following:

The mother who has no twins has time to relax.

This sentence makes a positive claim (as distinct from the negative one of the previous example) in that it says that there actually are people who do have time to relax, namely those mothers who have no twins. (Moreover, it does not explicitly say what happens to mothers of twins.) This seems to be a situation where synthesized attributes outperform global variables, since the rule at the relative clause level can simply refuse to pass on the negation marker.

Negation is not the only such troublemaker. Arranging subject-verb, adjective-noun and determiner-noun agreement also requires passing information around the tree, especially when translating into French, where word-for-word translation does not necessarily result in correct agreement. Again, having more than one clause makes difficult the use of global variables, particular when a plural relative clause is separating a singular subject from its verb. Consider the five subject-verb agreements in:

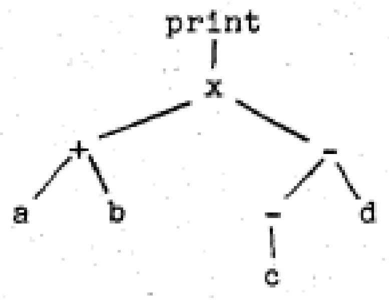
> As I walked into the saloon, the three men whom Jim talked to after I left him yesterday got up and slowly walked towards me.

All of these problems are "marker" type problems. Even worse is passing stylistic information from a word at the bottom of a tree to a clause node higher up, where this information is to be used to alter the whole structure of the translated clause. Again it is important that the appropriate clause get this information.

The mechanism we want here is that of the <u>local variable</u>, whose scope is the clause with which it is associated. With many clauses we will associate many more local variables corresponding to the various markers and other messages that each clause may want. Similarly, we will associate other local variables with noun phrases, to achieve adjective-noun and determiner-noun agreement. In the case of the subject, some of these markers (person and number, but not gender) must be shared with the clause as well, to ensure subject-verb agreement, but we do not want the clause to share the object's variables. Also, a relative clause such as "who sleeps" needs the same information from its govenor as does the principal clause. Moreover, we will want to pass not only markers, but also word-specific programs written at the dictionary level but intended for use at the clause or other level. (Winograd makes use of this technique for putting the right programs in the right places.) The implementation of local variables must be able to handle these combinations.

The first version of LINGOL implemented all of this in an unimaginative and not very general way. Eventually, we saw the light and came up with the program paradigm for syntax-directed translation.

The program paradigm says that the surface structure tree is a program. At each node of the tree there is a function, and the subtrees of that node are the arguments of that function. For example, if we have a tree labelled



this corresponds to the program "print (a + b)x((-c)-d)".

Since LISP has a mechanism for local variables (two,in fact - PROG variables and LAMBDA variables), by adopting the program paradigm we automatically get local variables. Moreover, because we can write the code for each function separately, we attain a very high level of modularity, which we have found pays off handsomely when one tries to add new rules to an already operational LINGOL program.

The mechanism we use for running these programs differs slightly from LISP's usual EVAL operator. The main difference is that it evaluates the function at each node first, giving the function the responsibility for evaluating subtrees at its leisure, and controlling the scopes of variables for different subtrees.

To illustrate all of this, we shall develop a small French translator. Imagine we are seated at a computer console. The following session has all typing errors and stupid mistakes edited out, since they rapidly become boring and obscure the main issues.

First we load the system.

```
L↑K!

LISP 229CX
ALLOC?
READY-TO-READ-GRAMMAR:
```

We are now talking to LISP. To get to talk to LINGOL, type (HELLO).

```
(HELLO)
FIRST ENTER YOUR GRAMMAR. THEN SAY HELLO AGAIN.
TOP-LEVEL:
```

So LINGOL is not yet educated. We could tell LISP to read in our dictionary and grammar from a file, but since we don't have such a file we will simply type it in at the terminal.

First, let us give LISP a few words.

```
(DICTIONARY)
(THE DET 0 '(LE))
(DOG NOUN 0 '(CHIEN))
(SEA NOUN 0 '(MER))
(LOVE VERB 0 '(AIM))
()
DICTIONARY-IN
```

Each entry has four items, a word, its part of speech, the cognitive program and the generative program.

The cognitive part is a LISP s-expression (or program) that should evaluate to a number to indicate to LINGOL our satisfaction or otherwise with this choice of interpretation for this word. It is relevant only when a given word has two dictionary entries corresponding to two parts of speech. Under these circumstances, we might write a program for each entry to inspect the environment to see how reasonable the corresponding interpretation is. These programs would be executed if and when both interpretations were found to make sense given the context to the left, e.g., it would be executed in "the scout flies..." but not in "the big flies...", where "flies" is listed as both a noun and a verb. This component of the entry need not concern us further here; we will remain neutral by writing 0 everywhere, unless we happen to dislike the entry itself, in which case we will write -1, or -2 if we are in a bad mood.

The generative part is a function destined to be tacked onto the surface structure. Since words are at the leaves of the tree, they have no arguments. In the case of "the", when the tree is evaluated, the corresponding leaf will return a list of one element (LE) as its value. The symbol ′ is a quotation mark, and means "literally", so as LINGOL will not think (LE) is a program to be executed. The other entries are all similarly structured. The reason we use a list of one word rather than the word itself is that we are going to APPEND these lists together to form longer lists.

Now we want a grammar to make sense out of the words in combination.

```
(GRAMMAR)
(SENTENCE (NP PRED) 0 (REPLY (APPEND !L !R) CHAR))
(NP (DET NP) 0 (APPEND !L !R))
(NP NOUN 0 !D)
(PRED (VERB NP) 0 (APPEND !L !R))
()
GRAMMAR-IN
```

Each rule is of the form (LEFT RIGHT COG GEN). The first two items should be interpreted as a context-free rule LEFT $\rightarrow$ RIGHT, where RIGHT is either one category or a list of them if more are needed. At present LINGOL only permits RIGHT to have at most two categories; to get more, one should use extra names and rules in the standard way.

The item COG is exactly as for the corresponding dictionary item, except that it may be invoked for more complex types of ambiguity, usually structural. As with the dictionary, we shall write no non-trivial programs here, although we may occasionally use a negative number when we write a rule which we do not expect to need very often.

The item GEN is a more complex item than its dictionary counterpart, since it can take arguments, which are written !D (down) if RIGHT is a syntactic category, and !L (left) or !R (right) if RIGHT is a list of two categories. These are not variables but programs which run the program for the corresponding subtree.

The first rule takes the translation of the NP and the PRED and appends them into a single list. For example, if the NP were (LE CHIEN) and the PRED were (AIME LE MER), then (APPEND !L !R) would produce (LE CHIEN AIME LE MER). The function (REPLY L T) is a LINGOL function which allows the generative phase to type out on the console the words in L, followed by the value of T. The variable CHAR is a LINGOL variable which makes available to the generative phase the character used to terminate the input string. (In the near future we shall give this to the cognitive phase instead, where it belongs.) In this case, we simply echo CHAR back to the console.

The other rules are similar, but without the REPLY. Hopefully they are all self-explanatory.

Let us try again to start LINGOL.

```
(HELLO)
TYPE SENTENCE FOLLOWED BY . ! OR ?
```

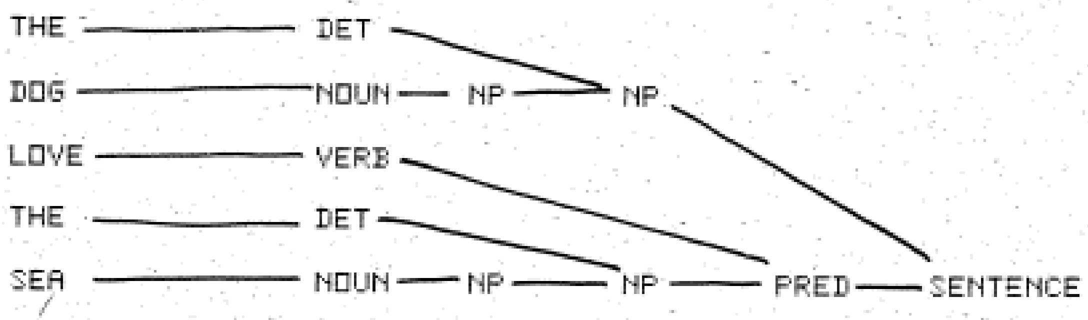
So far so good. Now for some sentences.

```
DOG LOVE SEA.
CHIEN AIM MER.
242. MILLISECONDS.

THE DOG LOVE THE SEA.
LE CHIEN AIM LE MER.
391. MILLISECONDS.

THE SEA LOVE DOG?
LE MER AIM CHIEN?
317. MILLISECONDS.
```

There is nothing to say here except to comment on the timing. This includes reading in the sentence and performing morphemic analysis (a feature to be described later), requiring about 30 milliseconds per word, or more if it is not in the dictionary. Parsing takes from 50 to 100 msecs per word depending on the complexity of the surface structure being produced, rather than on the size of the grammar. Parsing speed is essentially linear in the number of words in the sentence, given a reasonably intelligently written grammar of English. The timing of the generative phase varies enormously, as a function of the complexity of the user's semantic programs. In these examples we are probably spending about 10 milliseconds per word. The slowness is due to LINGOL's being written in LISP.

It would be nice if we could inspect the tree on which we are operating. We can do this by telling LISP to set the flag TREE. To get LINGOL to pass on a message to LISP, precede it with a slash.

```
/(SETQ TREE T)
T
THE DOG LOVE THE SEA.
```



(We have filled in the lines to show the connections.) This device is one of several debugging aids. It is also possible to monitor the activity of the parser as it discovers phrases, to see why it is not finding the right ones. The start and end positions of each discovered phrase are given, along with the rule used to discover it.

```
/(SETQ SHOWFOUND T)
T
THE
1. 1. DET THE
DOG
2. 2. NOUN DOG
2. 2. NP NOUN
1. 2. NP (DET . NP)
LOVE
3. 3. VERB LOVE
THE
4. 4. DET THE
SEA
5. 5. NOUN SEA
5. 5. NP NOUN
4. 5. NP (DET . NP)
3. 5. PRED (VERB . NP)
1. 5. SENTENCE (NP . PRED)
.
LE CHIEN AIM LE MER.
682. MILLISECONDS.
```

We can also watch the EVAL mechanism for the semantics (called SEVAL) returning values up the tree, with the help of the LISP debugging aid TRACE:

```
/(TRACE (SEVAL VALUE))
(SEVAL)
THE DOG LOVE THE SEA.

(1. ENTER SEVAL)
(2. ENTER SEVAL)
(3. ENTER SEVAL)
(3. EXIT SEVAL (LE))
(3. ENTER SEVAL)
(4. ENTER SEVAL)
(4. EXIT SEVAL (CHIEN))
(3. EXIT SEVAL (CHIEN))
(2. EXIT SEVAL (LE CHIEN))
(2. ENTER SEVAL)
(3. ENTER SEVAL)
(3. EXIT SEVAL (AIM))
(3. ENTER SEVAL)
(4. ENTER SEVAL)
(4. EXIT SEVAL (LE))
(4. ENTER SEVAL)
(5. ENTER SEVAL)
(5. EXIT SEVAL (MER))
(4. EXIT SEVAL (MER))
(3. EXIT SEVAL (LE MER))
(2. EXIT SEVAL (AIM LE MER)) LE CHIEN AIM LE MER.
(1. EXIT SEVAL NIL)
925. MILLISECONDS.
```

The numbers indicate the depth in the parse tree (q.v. above). This routine is extremely helpful for verifying that all functions are producing the correct output, and also for discovering where in the tree SEVAL runs into trouble.

We are not yet ready to translate the Canadian Hansard. Let us put in a variable to denote gender. The appropriate scope for the variable is an NP, since gender does not affect the verb. We need to tell LISP to change the grammar (our grammar is not yet elaborate enough to get LINGOL to do this for us).

```
/<GRAMMAR>
(NP (DET NP) 0 ((LAMBDA (GEND) (APPEND !L !R)) 'M) )
()
GRAMMAR-IN
```

LISP will now have replaced our old rule with the new one. (Only the components LEFT and RIGHT are used to identify and delete the old rule.)

We have used LAMBDA rather than PROG to declare our new variable. Had we used PROG we would have said

```
(PROG (GEND) (SETQ GEND 'M) (RETURN (APPEND !L !R)))
```

By using LAMBDA we save a SETQ and a RETURN. This is handy when there are a lot of variables to be SETQ'd.

The scope of GEND is just the NP, i.e., those functions which are called directly or indirectly by !L and !R here. GEND is set to "M" (masculine) as the default value (to enable us to eliminate specifying it in the dictionary), and will retain this value throughout its scope unless some function lower on the tree changes it, which we arrange now.

```
/<DICTIONARY>
(SEA NOUN 0 (PROG2 (SETQ GEND 'F) '(MER)))
()
DICTIONARY-IN
```

PROG2 evaluates each of its arguments, but only returns the value of the second.

We still have no way of using this information. Suppose we want determiner-noun agreement.

```
/<DICTIONARY>
(THE DET 0 (CDR (ASSOC GEND '((M LE)(F LA)))))
()
DICTIONARY-IN
```

ASSOC is a LISP table-lookup function, and CDR deletes the indicator in the discovered table entry (recall that we want (LE), not LE).

Hopefully we will find that the sea is LA MER.

```
THE DOG LOVE THE SEA.
LE CHIEN AIM LE MER.
398. MILLISECONDS.
```

There is a problem here with timing - we are trying to test GEND before it is set. The fault can be corrected from the NP rule, by doing !R before !L, on the grounds that the noun will never have to consult the determiner. This can be done by first assigning !R to R. (For clarity, we revert to a PROG.)

```
/(GRAMMAR)
(NP (DET NP) 0 (PROG (GEND R)
                     (SETQ GEND 'M)
                     (SETQ R !R)
                     (RETURN (APPEND !L R))))
()
GRAMMAR-IN
```

Now we can try again.

```
THE DOG LOVE THE SEA.
LE CHIEN AIM LA MER.
436. MILLISECONDS.

THE SEA LOVE THE DOG.
LA MER AIM LE CHIEN.
436. MILLISECONDS.
```

So it now seems to work. Had both the DET and the NOUN depended on one other for various features, instead of doing one before the other we would have ignored the order and done the appropriate table lookup higher up in the tree - the dictionary would simply have passed the whole table up instead of doing the lookup itself. This would work because the table lookup would be carried out with "complete information", i.e., after both !L and !R had terminated execution.

Verb conjugation seems to be next.

```
/(GRAMMAR)
(SENTENCE (NP PRED) 0
  ((LAMBDA (PERSON NO)
           (REPLY (APPEND !L !R) CHAR))
   3 '(SING))
()
GRAMMAR-IN
```

The default value for PERSON is 3, and for NO it is SING. To use these variables we need some dictionary entries.

```
/(DICTIONARY)
(LOVE VERB 0
  (LIST (COUNT (CDR (ASSOC NO
                           '((SING AIME AIMES AIME)
                             (PLUR AIMONS AIMEZ AIMENT))))
               PERSON)))
(I NOUN 0 (PROG2 (SETQ PERSON 1) '(JE)))
(YOU NOUN 0 (PROG2 (SETQ PERSON 2) '(TU)))
()
DICTIONARY-IN
```

(COUNT L N) yields the Nth element of L.

This now gives:

```
THE DOG LOVE THE SEA.
LE CHIEN AIME LA MER.
448. MILLISECONDS.

YOU LOVE THE SEA.
TU AIMES LA MER.
363. MILLISECONDS.
```

It seems silly to have to write so much in a dictionary entry for a regular verb. Why not just have a function REG which adds the right ending to the stem? We will define it by using DEFUN.

```
/(DEFUN REG (STEM)
  (LIST (CAT STEM (COUNT (CDR (ASSOC ND
                                     '((SING E ES E)
                                       (PLUR ONS EZ ENT))))
                             PERSON))))
REG
```

(CAT concatenates two strings.)

Now we can make most effective use of it.

```
/(DICTIONARY)
(LOVE VERB 0 (REG 'AIM))
(HUNT VERB 0 (REG 'CHASS))
(HIT VERB 0 (REG 'FRAPP))
(CLIMB VERB 0 (REG 'MONT))
()
DICTIONARY-IN
```

We can still enter irregular verbs the old way, or better, we can define another function whose six arguments are the six conjugations. (We resist the temptation.)

It would be nice to be able to distinguish singular and plural. This raises the morphological problem of detecting an "s" at the end of plural words. One solution is to make a dictionary entry for each plural word. But we can do better than this. LINGOL allows the user to identify suffixes by saying

```
/(DEFPROP S T SUFFIX)
S
```

This says that it is True that S is a SUFFIX. The same can be done for PREFIX. If LINGOL fails to find a word in the dictionary, it tries to remove a suffix. If it succeeds, it looks in the dictionary for the stem. For as long as it keeps failing to find anything in the dictionary it keeps removing suffixes, and after that prefixes. When it is done the effect is as if the original word had been made several words, e.g., UNKINDNESSES becomes UN KIND NESS ES if UN is a prefix and NESS and ES are suffixes. All information that these words were once one word is discarded, which could conceivably create unwanted ambiguities, although it seems unlikely for most affixes.

The word is eventually reassembled by the user's grammar, e.g.,

```
/(GRAMMAR)
(NOUN (NOUN S) 0 (PROG2 (SETQ NO 'PLUR) (LIST (CAT (CAR !L) 'S)))
()
GRAMMAR-IN
/(DICTIONARY)
(S S 0 0)
()
DICTIONARY-IN
```

```
THE DOGS LOVE THE SEA.
LE CHIENS AIMENT LA MER.
712. MILLISECONDS.
```

Ah, the determiner rule is no longer valid.

```
/(DICTIONARY)
(THE DET 0 (COND ((EQ NO 'PLUR) '(LES))
                 ((CDR (ASSOC GEND '((M LE)(F LA)))))))
()
DICTIONARY-IN

THE DOGS LOVE THE SEA.
LES CHIENS AIMENT LES MER.
918. MILLISECONDS.
```

Also we need a new PERSON and NO for the object, although GENDER is all right because it is in the NP rule.

```
/(GRAMMAR)
(PRED (VERB NP) 0 (APPEND !L ((LAMBDA (PERSON NO) !R) 3 'SING)))
()
GRAMMAR-IN
```

```
THE DOGS LOVE THE SEA.
LES CHIENS AIMENT LA MER.
561. MILLISECONDS.
```

The reason we keep finding errors is because we are writing the program as though we were beginners. With a little experience, the user can learn to anticipate most of these problems at the start.

This scheme has the advantage that the user is not constrained to any one morphological system, but can write his own in the same language as he writes his semantics. It has another advantage in that morphological processing can be interleaved with semantic processing. For example, when LINGOL gives up on a word altogether, it assigns it the category UNKNOWN and supplies the word in the generative phase. If we want to implement Thorne's (1968) closed-class dictionary, inwhich unknown words are parsed as nouns, verbs or adjectives depending on which interpretation makes the best syntactic sense, then we could write rules such as

```
/(GRAMMAR)
(NOUN UNKNOWN 0 (LIST !D))
(VERB UNKNOWN 0 (REG !D))
()
GRAMMAR-IN
```

```
THE DOGS PREFER THE CATS.
LES CHIENS PREFERENT LES CATS.
787. MILLISECONDS.
```

Notice how the issue of deciding what part of speech the word is is dealt with independently of, e.g., making "CAT" plural. Also notice that the parser correctly guessed the parts of speech, and went on to conjugate "correctly" the unknown verb. However, "cats" is a bit of an Anglicism. Our program is starting to look quite clever already without our having done very much to it yet. We have only seven grammar rules, one function (REG) and a few dictionary entries.

In the example of Figure 5 (section 1), the rules involving UNKNOWN have for their generative component a program that queries the user about the translation.

These examples could go on indefinitely. To see what can be achieved with a few more hours work, refer back to Figure 5. That example still has very little grammar - approximately twenty rules. However, it has a page of LISP functions for doing liason, various agreements, and handling tricky things like LES versus DES in the object position.

These examples bring this section to an end. There is no section 3.3 on Pragmatics - this is entirely the user's problem. Figure 3 (section 1) gives examples from a LINGOL program in which the user successfully interfaced his semantics to quite non-trivial pragmatics. It is not yet clear whether LINGOL should ever address pragmatic issues.

4. Conclusions

We have described a programming language for natural language processing programs. We discussed the reasons for each of the major design decisions. We presented a session with the system in which we developed a trivial fragment of an English-to-French translator. With adequate imagination, the reader should be able to project at least some of the potential of LINGOL. What may be more difficult to see are the present limitations of the system.

We have already suggested that our separation of semantics from the syntax does not present serious problems. Whether this is true we leave to further experiments with LINGOL. It should be noted that LINGOL is still in its infancy; so far the author has invested approximately three months' work in it, over the two and a half years of its existence.

At present, conjunction is not handled at all by LINGOL, except in so far as one may supply context-free rules for each syntactic category to be conjoined (which is most). This is tedious at best, and is not even always possible. One wants to deal not only with "The Chinese have short names and the Japanese long" but with "He eloped with and married the farmer's daughter." Neither of these are at all well handled by context-free grammars, regardless of what we write in the cognitive component of our rules. Winograd's system deals with these sorts of problems simply by being more procedure-oriented. This provides the necessary flexibility to deal with pathological cases.

Another difficult area is that of adverbs, which may appear in many places in a sentence, but which always modify the verb of the clause they appear in (unless they modify an adjective). It should not be necessary to give rules for each of the places an adverb may appear. It suffices to rely mainly on semantic connections to establish the role of the adverb, and this is one place where concept structures (Shank 1970) are of value.

Both of these problems will be studied in the near future, to see how best to change LINGOL to deal with them without losing the attractive programming convenience afforded by context-free rules in conjunction with LISP semantics. In the meantime, the system as it stands at present is available from the author for experimental use. A LISP environment is required, with at least 20K words of memory. An obvious application for LINGOL is as a pedagogical tool in a computational linguistics course, for introducing students painlessly to one method of writing actual programs that do something useful with English other than parsing it for the sake of the parse tree. We have used it for this purpose during the Independent Activities Period at MIT this January. One student wrote an English-to-unpointed-Hebrew translator! We ask only that users keep us up-to-date with the uses to which they put LINGOL.

Bibliography

Aho, A.V. and J. Ullman, 1972. The Theory of Parsing, Translation and Compiling, Vol. 1, Prentice-Hall, Inc., New Jersey.

Bobrow, D.G., 1964. "METEOR - A LIST Interpreter for String Transformations," in Berkeley, E.D. and D.G. Bobrow (eds.) The Programming Language LISP: Its Operation and Application, Information International, Inc. Cambridge, Massachusetts.

Bobrow, D.G. and J. B. Frazer, 1969. "An Augmented State Transition Network Analysis Procedure," Proceedings of IJCAI, 1969, 557-568.

Charniak, E.C., 1972. "Toward a Model of Children's Story Comprehension," AI TR-266, MIT, Cambridge, Massachusetts.

Farber, D.J., R.E. Griswold and I.P. Polonsky, 1964. "SNOBOL, A String Manipulation Language," Journal of the ACM, 11, 2, 21-30.

Fillmore, C.J., 1968. "The Case for Case," in Bach and Harms (eds.) Universals in Linguistic Theory, Holt, Rinehart & Winston, 1-90.

Green, P.F., A.K. Wolf, C. Chomsky and K. Laugherty, 1961. "BASEBALL: An Automatic Question Answerer," in Feigenbaum and Feldman (eds.) Computers And Thought, 207-216.

Hays, D.G., 1964. "Dependency Theory: A Formalism and some Observations," Language, 40, 4, 511-525.

Katz, J.J. and J. A. Fodor, 1964. "The Structure of a Semantic Theory," in Fodor and Katz (eds.), The Structure of Language, 479-518.

Klima, E., 1964. "Negation in English," in Fodor and Katz (eds.) The Structure of Language, 246-323.

Knuth, D.E., 1968. "Semantics of Context-Free Languages," Math Systems Theory, 2, 127-145.

Lewis P.M. and R. E. Stearns, 1968. "Syntax-directed Transduction," Journal of the ACM, 15, 3, 465-488.

Narasimhan, R., 1969. "Computer Simulation of Natural Language Behavior," Invited paper, Conference on Picture Proc. Mach., Canberra, Australia.

Schank, R., L. Tesler and S. Weber, 1970. "Spinoza II - Conceptual Case Based Natural Language Analysis," AI-109, Stanford University, Stanford, California.

Simmons, R.F., S. Klein and K. McConlogue, 1964. "Indexing and Dependency Logic for Answering English Questions, Amer. Doc., 15, 3, 196.

Thorne, J., P. Bratley and H. Dewar, 1968. "The Syntactic Analysis of English by Machine," in Michie, D. (ed.) Machine Intelligence 3.

Warshall, S. and R. M. Shapiro, 1964. "A general purpose table driven compiler," Proc. AFIPS SJCC, 25, 59-65. Spartan, New York.

Winograd, T., 1971. "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," Project MAC TR-84, MIT, Cambridge, Massachusetts.

Woods, W.A., 1967. "Semantics for a Question-Answering System," Report no. NSF-19, the NSF, Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts.

Woods, W.A., 1969. "Augmented Transition Networks for Natural Language Analysis," Report No. CS-1 to the NSF, Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts.

Yngve, V.H., 1963. "COMIT," Communications of the ACM, 6, 3, 83-84.