TR 273 T

31

Coling 76

A Multi-Processing Model for Natural Language Understanding

R. Smith and F. Rawson,
Institute for Mathematical Studies in the Scial Sciences,
Stanford University,
Stanford, California 94305.

A Multi-Processing Model of Natural Language Understanding[1]

Robert Smith
Stanford University

Freeman L. Rawson
IBM Corporation

1 Introduction

One of the problems in many natural language understanding systems is the complexity of the programs that generate the internal representations of the input sentences. To simplify these programs, we suggest the use of multi-processing control structures. We outline an extension to LISP called PLISP that will allow semantic functions to be separate processes running concurrently. To illustrate the properties of PLISP, we give several examples.

Most of the examples are drawn from the language of set theory, and the intended goal of the understanding process is to produce a first-order-like representation of a statement of mathematics for use in a proof-checking system.[2] This paper does not directly concern issues of knowledge representation; instead it is directed to the issue of the control structures that facilitate the processing of surface-level syntax into any particular representational schema that may be appropriate. We especially want to show how semantic and contextual information affects the flow of control, and to illustrate this we focus on the roles of quantifiers and operators.

2 Informal Mathematical Language

Understanding informal mathematical English requires the ability to determine the correct scopes of quantifiers and other operator-like words and phrases. Consider the following sentence:

(1) If there are three sets such that the first is a subset of the second and the second is a subset of the third, then the first is a subset of the third.

Using a phrase-structure grammar constructed mainly on syntactic grounds we might obtain the tree given in Figure 1.

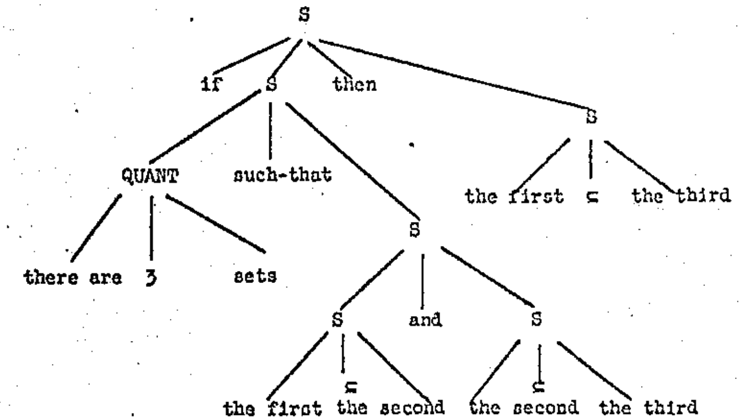

Figure 1. Surface Syntax Tree

The "logical form" of (1) (i.e., the first-order representation that we hope to determine) is similar to:

[1]This research was supported by the National Science Foundation under NSF Grant EPP 74-15016, A01. The opinions expressed in this paper are those of the authors and not of the National Science Foundation, Stanford University, or the International Business Machines Corporation.

[2]The proof checker is described in [8].

(2) $(\forall x,y,z)$[IF x is a set and y is a set and z is a set and $x \subseteq y$ and $y \subseteq z$ THEN $x \subseteq z$]

which has the schematic form:

(3) $(\forall$ <variable-list>$)(\varphi \Rightarrow \Psi)$

The straightforward reading of the tree in Figure 1 is more likely to yield:

(4) $[(\exists$ <variable-list>$)\varphi] \Rightarrow \Psi$

Logical form (4) cannot be correct; as seen in (2), the quantification actually governs the entire sentence, and its sense is not existential (as the phrase "there is" would indicate) but instead is universal[3].

One standard solution to this problem is to rewrite the phrase-structure grammar to produce a "flatter tree" that puts the quantifiers and operators nearer the surface, bringing the components of the sentence into the view of more global functions for resolving scope and anaphora.[4] This often creates a larger grammar than necessary with some duplication of production rules that cover syntactically similar cases. Also, it is more computationally natural to represent the information about scopes of quantifiers in the evaluation functions than to attempt to embed the quantifier scope rules in the syntactic component. This is not a new observation, but it is one that is often ignored in traditional linguistics.

[3] Note that $(\forall x)(\varphi \Rightarrow \Psi)$ is logically equivalent to $[(\exists x)\varphi] \Rightarrow \Psi$ just in case <u>x is not free in</u> Ψ.

[4] Winograd's early work [11] tends to this solution in some cases.

Standard surface-level grammars have a certain psycholinguistic appeal, both for their simplicity and their correspondence to classical conceptions of grammar. Thus, the tree in Figure 1 satisfies some intuitions about syntax. Unfortunately, it is not also an adequate representation of the semantics of the input sentence. The underlying structure--the control structure of the program needed to evaluate the meaning--is quite different from the surface structure of the sentence. The pronouns "first", "second", and "third" must refer to each of the "three sets"; further, the resolution of these pronominal references changes the scope and sense of the quantification: "there is" must actually govern the whole sentence, and that quantification is universal. This analysis supposes that the crucial computations--associated with the pronouns--have some access to the other computations, but the syntactically-motivated tree in Figure 1 constrains these computations too locally. It is intuitively clear that the right information is not available at the right time during the evaluation.

3 PLISP: Process LISP

We propose to evaluate the meaning of such sentences by associating a separate process with each node of the surface-level syntax tree; the original surface syntax corresponds to a <u>default control structure</u> for the evaluation. These semantic processes will run concurrently under a timesharing scheduler and will be able to poll the status of other processes and wait for them to finish; they can also alter the flow of control.

The language PLISP (for Process-LISP) is a theoretical extension to LISP, in much the same sense that CONNIVER [5] and QA4 [7] are also extensions to LISP. We plan to implement PLISP in INTERLISP-10 [10]. To define PLISP we must:

1. Give the data structures and environment that are provided by the interpreter.

2. Define the PEVAL function, which replaces EVAL as the main function-calling mechanism in PLISP.

3. Describe the primitive functions provided by PLISP and their effects on the PLISP environment.

PEVAL works like EVAL on standard LISP function types such as EXPR. Functions of type PEXPR, which are to create processes, are applied to their arguments by PAPPLY. PAPPLY takes as arguments a function P of type PEXPR and a list of arguments, and builds a new process, binding arguments of the new process and associating the code of the PEXPR P to the process. This creates a process activation record. The process is then put into the queue of runnable processes and run by the PLISP scheduler, which is a part of the PLISP interpreter. PAPPLY then returns a pointer to the process activation record. It should be noted that the arguments to a PEXPR may themselves be (pointers to) other processes. It is crucial to the concept of PLISP that the order in which information is "really" processed is not necessarily determined by PEVAL; instead, PEVAL creates a network of processes, called a dependency graph. A top-level form is said to have completed when the top-level process completes, at which point any pending processes will be automatically destroyed.

An attempt by a process to access an argument that is not "ready" --i.e., that has not been returned by the process computing it--will result in the accessing process faulting (and being blocked from computation) until the argument is available. This is analogous to an "I/O wait" status for a job in a time-sharing system.[5]

A number of primitive functions are available to examine and alter the flow of control and information in the evaluation. We shall describe how these are useful in natural language processing.

3.1 PLISP Activation Records

The process activation record is the data structure that defines a process and the relationship of that process to other processes.[6] As shown in Figure 2, the process activation record contains the following entries:

PROGRAM!TEXT is the name of the PEXPR function associated with the process. This field serves as a pointer to the PLISP expression that defines the PEXPR.

PARAMETERi is the ith parameter to the process, which may be either another process from which values are expected or a value. There is one PARAMETERi entry for each i between 1 and n, where n is the number of arguments accepted by the PEXPR.

PARAMETER!NAMEi is the name of the ith parameter.

PARAMETER!STATUSi contains the status of parameter i. If the parameter is a value, then it contains the atom VALUE, else if the parameter points to a process P, then it contains the status of that process, which will be the PROCESS!FLAG entry of the activation record of that process.

[5]It is actually closer to "fork wait" status for the forks in the TENEX timesharing system [2]. TENEX forks correspond somewhat to our processes, except that inter-process communication and control is quite primitive in TENEX v. 1.33.

[6]These records are similar to those given by Lampson, Mitchell, and Satterthwaite [4], and to those of Bobrow and Wegbreit [1]; the idea of multiple attributes associated with the process is taken from Knuth [3].

PROCESS!FLAG contains the current status of the process. Possible values include:

1. RUNNING: The process has currently been selected to run by the PLISP scheduler.

2. READY: Currently, the process is not running, but it is runnable.

3. WAITING: The process is waiting for other process(es) to finish. WAITING!FOR!LIST is a list of those processes.

4. COMPLETED: The process is no longer runnable, and the VALUE cell of the process activation record has an explicit value.

5. TERMINATED: A process P is marked TERMINATED when a PLISP TERMINATE has been issued against P, but P does not yet have an explicit value. Instead the VALUE cell of the process activation record points to another process which, upon completion, will provide the VALUE for the terminated process. A TERMINATED process cannot be resumed.

WAITING!FOR!LIST is the list of processes upon which the current process is waiting.

DEPENDENCY!LIST is the list of processes that are waiting on the current process.

VALUE is the value returned by the process when it has COMPLETED. If the process is TERMINATED, then the VALUE cell contains a pointer to the process that will, upon its completion, supply the value for the current process. If the process is neither TERMINATED nor COMPLETED, the contents of VALUE are undefined.

ATTRIBUTE!LIST is a list of attributes/value pairs associated with the current process. These attributes are similar to those used by Knuth [3]. VALUE, described above, is a special attribute distinguished from others in that a process returns its VALUE attribute.

Process Activation Record	
PROGRAM!TEXT	(LAMBDA ...)
PARAMETER1	
PARAMETER!NAME1	
PARAMETER!STATUS1	
other parameters	
PROCESS!FLAG	
WAITING!FOR!LIST	list of processes upon which waiting
DEPENDENCY!LIST	list of processes waiting for this
VALUE	main value returned by process
ATTRIBUTE!LIST	other attributes returned
storage	

Figure 2. Process Activation Record

We note that these activation records are fairly simple. The real implementation will contain additional information in the activation records much as in the "frame" scheme of Bobrow and Wegbreit [1].

3.2 Process Primitives

We now list the process manipulation primitives that are available in PLISP. The main primitives are:

value ← TOPPEVAL (<form>)

TOPPEVAL returns the value of the top-process of <form> and destroys all waiting lower-level processes. This is the function that is called to evaluate a sentence. TOPPEVAL calls PEVAL to do the actual evaluation and waits for the top-level process to complete before returning.

value ← PEVAL (<form>)

PEVAL returns the value of the <form> by using the PEVAL algorithm.

value ← PAPPLY $(P, v_1, \ldots, v_n)$

If P is a PEXPR, PAPPLY creates a process (i.e., a process activation record) with PROGRAM!TEXT P and arguments $v_1, \ldots, v_n$. The arguments may be processes. If P is any other type of function, PAPPLY is simply the standard LISP APPLY function.

Other functions, also available to the user, include:

Each element of the process activation record can be read or written using record/reference notation. Thus,

t ← PROGRAM!TEXT [P]

reads the name of the function associated with P.

v ← AWAIT (P)

The issuing process waits until P has finished and the value returned by P is returned as the value v of the AWAIT function. If P has already COMPLETED then the value originally returned by P is fetched from the VALUE cell of the activation record for P.

TERMINATE (P,<value>,<attribute assignment list>)

TERMINATE takes the following actions, which are done simultaneously: 1) assigns <value> to VALUE[P]; 2) makes all the assignments in <attribute assignment list>; 3) sets STATUS[P] to TERMINATED if <value> is a process, or COMPLETED if <value> is an explicit LISP value, and returns the value <value> to any processes in DEPENDENCY!LIST[P]; and 4) P is removed from the DEPENDENCY!LIST of any process on the WAITING!FOR!LIST of P. In other words, P is forced to return <value> as a value, and additional attributes of P may also be assigned, with other book-keeping operations performed.

PRETURN(<value>,<attribute assignment list>)

This is equivalent to

TERMINATE(self,<value>,<attribute assignment list>)

where self is the current process.

status ← ARGUMENT!STATUS (<argument>)

This is used, within a given function, to determine the status of a named argument to that function. If <argument> is a value, then it returns the atom VALUE; if <argument> is a process P, then it returns the status of P. If an argument to a process is another process, then directly reading that argument (e.g., in an assignment statement) will cause the reading process to fault; ARGUMENT!STATUS allows one to check first.

P ← SPLICE(Q,<argno>,<function>)

SPLICE inserts a process between the <argno> argument of process Q and whatever that argument points to. The third argument, <function>, which may be either the name of a PEXPR or a LAMBDA-expression (of one argument), is used for the PROGRAM!TEXT of this process. The created process, P, is returned as the value of the call.

4 Applying PLISP to Natural Language

We now consider several sentences, from the domain of elementary mathematics, that illustrate the features of PLISP in natural language processing. Elementary mathematics has fewer representational problems than, say, a dialogue about a child's birthday party; these examples do however show how PLISP processes can integrate semantic and contextual information in determining the scopes of operators and quantifiers.

Our sample PLISP functions for these sentences are written in an ALGOL-like meta-LISP syntax. These functions depend on some simple LISP functions that exist in our proof checker (see [8]) as well as some utility PLISP functions.

4.1 An Example of Noun Phrases

Consider the noun phrases:

(5) odd and even numbers

(6) odd and prime numbers

as they occur in the sentences

(5′) All odd and even numbers are divisible by 1.

(6′) All odd and prime numbers are bigger than 2.

Both of these noun phrases might have the same parse tree as in Figure 3. Notice that the "and" is embedded within an adjective phrase construction (labelled ANDADP).

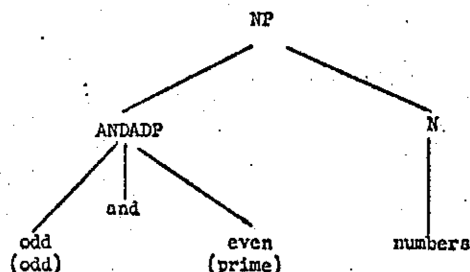

Figure 3. Noun Phrase Syntax Tree for (5) (and (6))

The semantic problems of the phrase result from the sense of the word "and". In phrase (5) the meaning is

(5″) "those numbers that are either odd or even",

and in phrase 2)

(6″) "those numbers that are both odd and prime".

One description of how we know this is that in (5) we are probably not talking about things that are both odd and even since there are none, hence we take the union of the sets involved as the meaning of "and", whereas in (6) the intersection interpretation is more plausible. This is not the only scenario possible; but let us show how to write PLISP functions that implement the above intuitions starting with the parse tree shown in Figure 3.

The top-level function will be FORM!INTERSECTION, trivially defined:

```
PEXPR FORM!INTERSECTION(LEFT,RIGHT);
RETURN(LIST('INTERSECTION,LEFT,RIGHT));
```

The function DECIDE!AND, associated with the node labelled ANDADP of Figure 3, is programmed as follows.

```
PEXPR DECIDE!AND(LEFT,RIGHT);
BEGIN NEW P;
COMMENT Find the process that is superior to the current process;
P ← FIND!SUPERIOR!PROCESS;
COMMENT Check to see if the superior process will be forming
      an intersection;
IF PROGRAM!TEXT[P]='FORM!INTERSECTION THEN
    BEGIN NEW SECOND!ARG;
        COMMENT Pick up the second argument to the above intersection
                function;
        SECOND!ARG ← PARAMETER2[P];
        COMMENT Check to see if all three will fit together in
                a reasonable intersection;
        IF NOT PLAUSIBLE!INTERSECTION(LEFT,RIGHT,SECOND!ARG) THEN
            BEGIN   COMMENT critical step;
                TERMINATE(P,PEVAL(
                  '(FORM!UNION(FORM!INTERSECTION LEFT SECOND!ARG)
                            (FORM!INTERSECTION RIGHT SECOND!ARG))));
                PRETURN(NIL);           COMMENT terminate self;
            END ELSE PRETURN(FORM!INTERSECTION(LEFT,RIGHT));
    END ELSE ... other cases ...;
END;
```

DECIDE!AND uses the utility function FORM!UNION that forms the union of two

sets, the function FIND!SUPERIOR!PROCESS that searches the pointers in the dependency graph within which the current process is embedded, and the function PLAUSIBLE!INTERSECTION. This last function decides whether or not a proposed intersection is "plausible" —for which one reasonable criterion is that the intersection be nonempty. The important thing is that the process mechanism allows us to write the DECIDE!AND function so that the critical semantic information (in this case, whether or not an intersection is "plausible") is used locally within the process associated with the word "and". In particular, there is no special code in the FORM!INTERSECTION function to handle this.

The different effects of DECIDE!AND can be seen by comparing Figures 4 and 5, which show dependency graphs for each of the phrases (5) and (6) at the critical points in the evaluation.

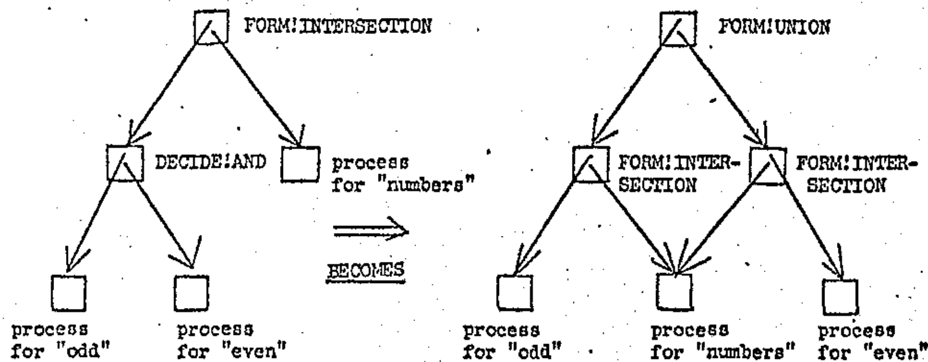

Figure 4. Dependency Graphs for (5)

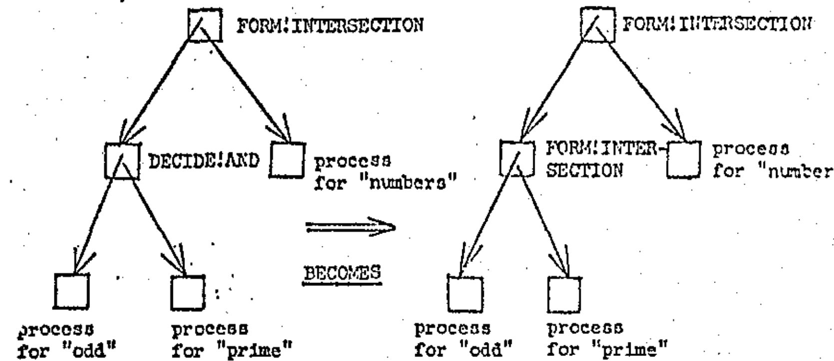

Figure 5. Dependency Graphs for (6)

In Figure 4, the DECIDE!AND process determines that the sense of the word "and" is <u>union</u>, and creates a new dependency graph based upon this heuristic. In Figure 5, where the sense of "and" is <u>intersection</u>, the original control structure is left essentially intact, using FORM!INTERSECTION in place of DECIDE!AND.

4.2 A Sentence Involving Quantification

The occurrence of two or more quantifiers together in a sentence, especially when they are embedded within inner phrases in the natural syntax, gives rise to the problem of determining the actual scopes of the quantifiers. Sometimes this is considered simply to indicate ambiguity, as in:

(7) Every man loves some woman.

which, it has been argued, has the two readings:

(7′) For every man, there is a woman (associated with that man) such that ...
(7″) There is a (single) woman such that for every man ...

We shall analyze two structurally similar sentences drawn from elementary mathematics.

(8) Every number is less than some number.
(9) Every natural number is greater than or equal to some natural number.

Sentence (8) probably has the logical form:

(8′) $(\forall x)(\exists y)(x < y)$

while (9), depending on the context, is probably an assertion of the existence of the number zero:

(9′) $(\exists y)(\forall x)(x \geq y)$

Both (8) and (9) may have the same surface-level syntax as shown in Figure 6.

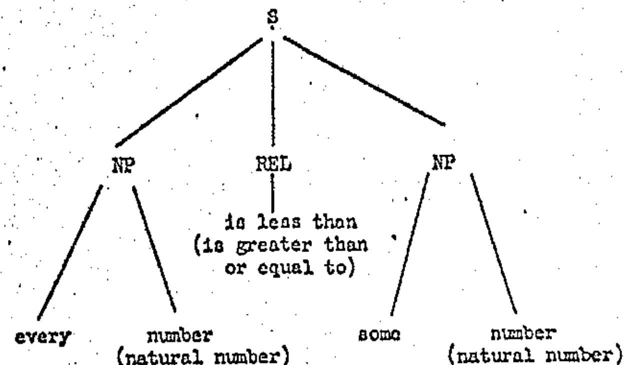

Figure 6. Syntax Tree for (8) (and (9))

We want to isolate the difficult code in the functions associated with the nodes that introduce the quantifiers. The other functions perform straightforward evaluations associated with their syntactic contexts: e.g., the top-level function is the same one that would handle the sentence

(10) $3 < 4$

by forming the appropriate first-order representation.

The quantifier nodes must return values consisting of the quantifiers themselves, any type assertions made by the quantificational phrase, and scope attributes indicating whether the quantification is over the whole sentence or some subpart.

We now give outlines of the critical PLISP functions for this example.

```
FEXPR FORM!REL(NP1,REL,NP2);
PRETURN(LIST(REL,NP1,NP2));
```

We note that this function does not have any special context-dependent code, thus meeting the criterion set out above.

```
PEXPR FORM!QUANT(QUANT,NP);
BEGIN
NEW I,Q,VAR;
COMMENT Search for other occurrences of the FORM!QUANT function to the
        left of the present one.  Wait for all such processes to
        terminate;
FOREACH I SUCHTHAT I is a process to the left of SELF  DO
   IF PROGRAM!TEXT[I]='FORM!QUANT THEN AWAIT(I);

COMMENT Generate a variable name and set the VALUE attribute for
        this process activation record.;
VAR ← FORM!NEW!VARIABLE;
COMMENT Create another process Q by splicing it into the dependency
        graph between S and T.  Begin by checking the nodes between
        T and S for another Q node.;
IF there is a process I between S and T THEN
    IF PROGRAM!TEXT[I]='QUANTFUNC THEN
    BEGIN
        COMMENT Compare the signs of the quantifiers;
        IF SIGN(QUANT) = SIGN(VALUE[I]) THEN
            BEGIN
                COMMENT The signs are the same and no
                        special ordering is required;
                SPLICE(I,1,'(LAMBDA (X) (QUANTFUNC
                                 X QUANT VAR ...type assertions...)));
            END ELSE
            BEGIN
                COMMENT The signs are different.  Hence, we must
                        decide where to splice this quantifier
                        into the structure.  This code assumes
                        that the universal is lexically first.;
                IF there is a unique Y such that for all X the
                   value of S is true THEN
                COMMENT Put the existential quantifier first.;
                SPLICE(T,1,'(LAMBDA (X) (QUANTFUNC
                                 X QUANT VAR ...type assertions...)))
                    ELSE
                COMMENT Put the universal quantifier first.;
                SPLICE(I,1,'(LAMBDA (X) (QUANTFUNC
                                 X QUANT VAR ...type assertions...)));
            END;
            COMMENT There is no intermediate node between S and T.;
    END ELSE SPLICE(T,1,'(LAMBDA (X) (QUANTFUNC
                               X QUANT VAR ...type assertions...)));
COMMENT This process evaluates to the variable we created;
PRETURN(VAR);
END;
```


The function QUANTFUNC creates a quantified formula, accepti arguments ARGNODE, SIGN, VARIABLE, TYPE!ASSERTION. It quantifies ove ARGNODE using the variable VARIABLE and a quantifier of type SIGN (e.g. existential or universal). The TYPE!ASSERTION is incorporated into th matrix. This is a standard operation in LISP programs dealing with logic.

Figures 7 and 8 depict the dependency graphs for the two sentences (8 and (9). In Figure 7 the semantic processor has ordered node Q1, th universal quantifier, above Q2, the existential quantifier, while th situation is reversed in Figure 8. Intuitively, the semantic processo determines the ordering by determining whether there is a unique y of th proper type such that for every x of the proper type, the matrix of th formula, represented by the node S, is true.

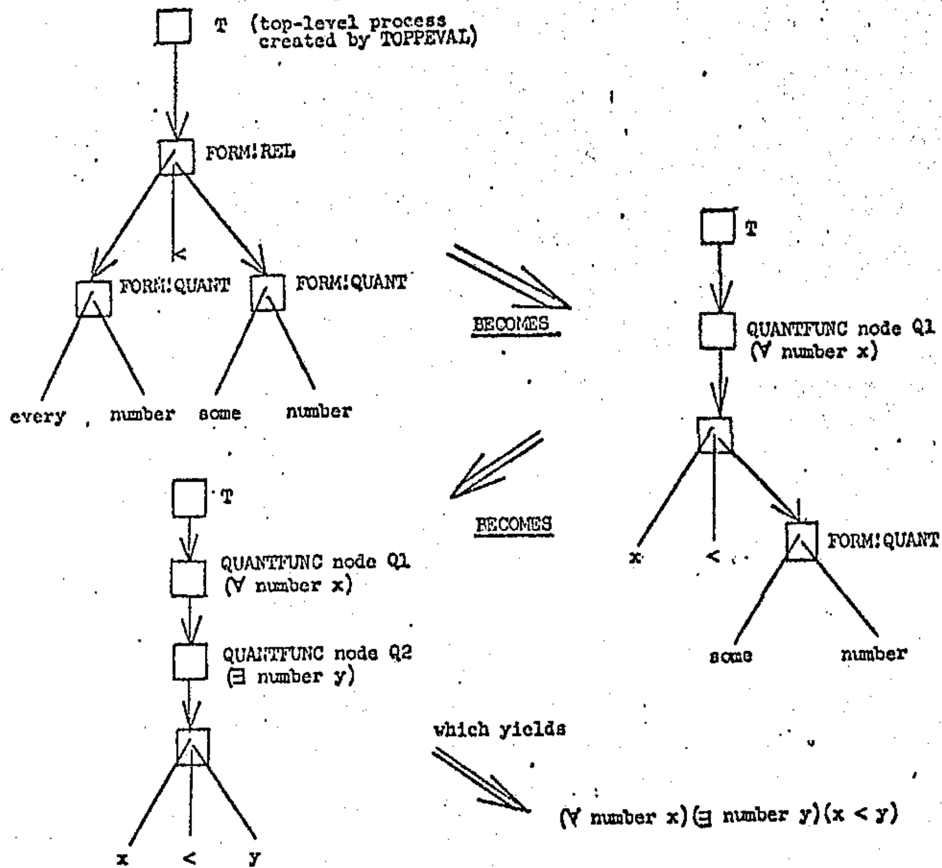

Figure 7. Dependency Graphs for (8)

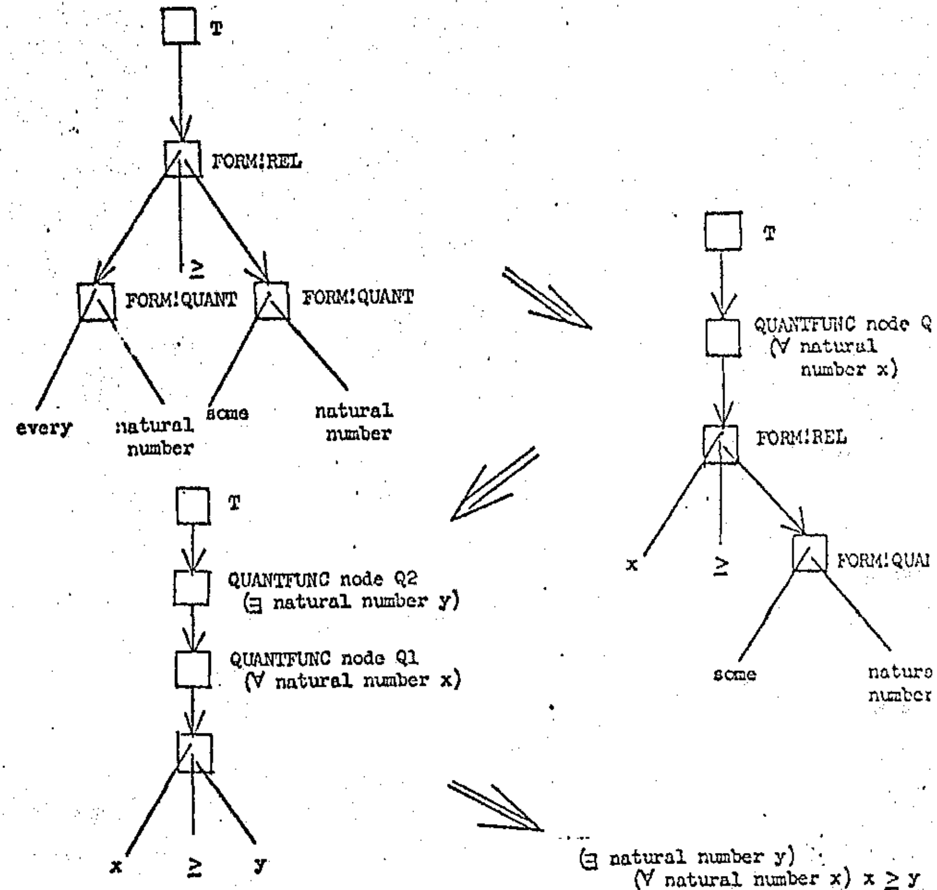

Figure 8. Dependency Graphs for (9)

4.3 Resolution of Scope and Anaphora

We now consider the more complex example given in the beginning of this paper:

(11) If there are three sets such that the first is a subset of the second and the second is a subset of the third, then the first is a subset of the third.

Figure 9 gives the syntax tree for (11).

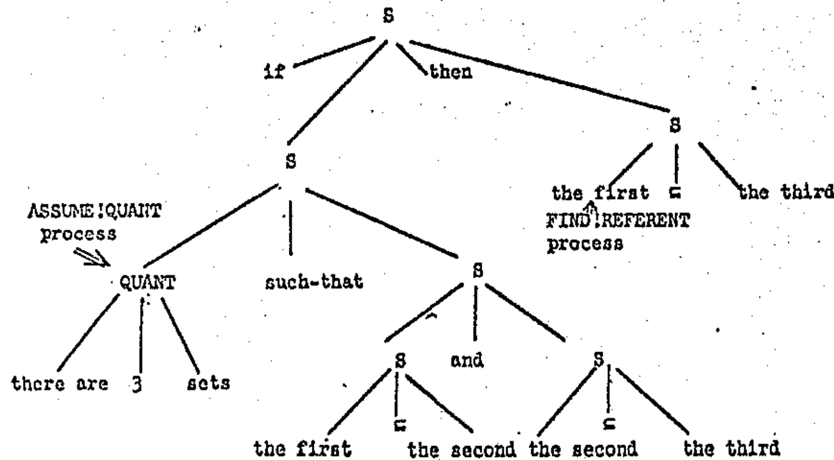

Figure 9. Surface Syntax for (11)

There are two critical semantic functions involved in understanding (11). The first, ASSUME!QUANT, handles the quantifier phrase "there are". It must create the variables X1, X2, X3, determine their types (i.e., they are all sets), and determine the sign and scope. The second important function, FIND!REFERENT, is associated with the pronouns "first", "second", and "third". These functions must find their referents, and check that any scope assumptions previously made are correct.

Below we outline the PLISP code for these two functions.


```
PEXPR ASSUME!QUANT(SIGN, NUMBER, TYPE);
BEGIN NEW I, VAR, VARLST, TYPELST, SIGNLST, SCOPELST;
COMMENT Sets the scope and value attributes.;
FOR I ← 1 STEP 1 UNTIL NUMBER DO
    BEGIN
        VAR ← FORM!NEW!VARIABLE;        COMMENT create a new variable;
        ... put VAR in VARLST ...
        ... assert that VAR has type TYPE and put in TYPELST ...
        ... set the sign in SIGNLST to SIGN ...
        ... set the scope in SCOPELST to 'LOCAL ...
    END;
COMMENT Now, return the quantified expression, which is simply built
        up by the appropriate logic-related functions setting some
        attributes of the current process simultaneously.;
PRETURN (quantified expression,ASSUMED!VARIABLES[SELF]   VARLST;
                               ASSUMED!SCOPE[SELF]   SCOPELST);
END;

PEXPR FIND!REFERENT(VARNAME);
BEGIN NEW I,V,P,NEWSIGN,VAR;
... first, we check to see if there is a reference, in
        processes to the left, and if we find one, we use it ...
COMMENT If no referent was found, wait for any likely processes
        to terminate. In particular, we will look for
        quantifier processes to terminate.;
FOREACH I SUCHTHAT I is a quantifier process to the left of SELF DO
    BEGIN
        V ← AWAIT(I);
        COMMENT check for a "match" between VARNAME and the variables
                created by process I.;
        IF match between one of the variables created by I and VARNAME
          THEN BEGIN
                VAR ← pick up that variable;
                IF variable is correctly in scope THEN PRETURN(VAR)
                ELSE BEGIN
                     COMMENT this is the critical part!;
                     P ← ... find the process above the process
                             that dominates the current scope
                             plus the original scope;
                     NEWSIGN ← find if we need to change the
                             sign of the quantifier;
                     SPLICE(P,argument!number,
                             '(LAMBDA (X) (QUANTFUNC VAR NEWSIGN
                                   ... copied type assertions)));
                     PRETURN(VAR);
                    END;
                END;
    END;
END;
```

Figure 10 gives the dependency graphs for (11) at crucial stages of the evaluation. The process FIND!REFERENT changes the scope and sense of the quantifier to govern the entire expression.

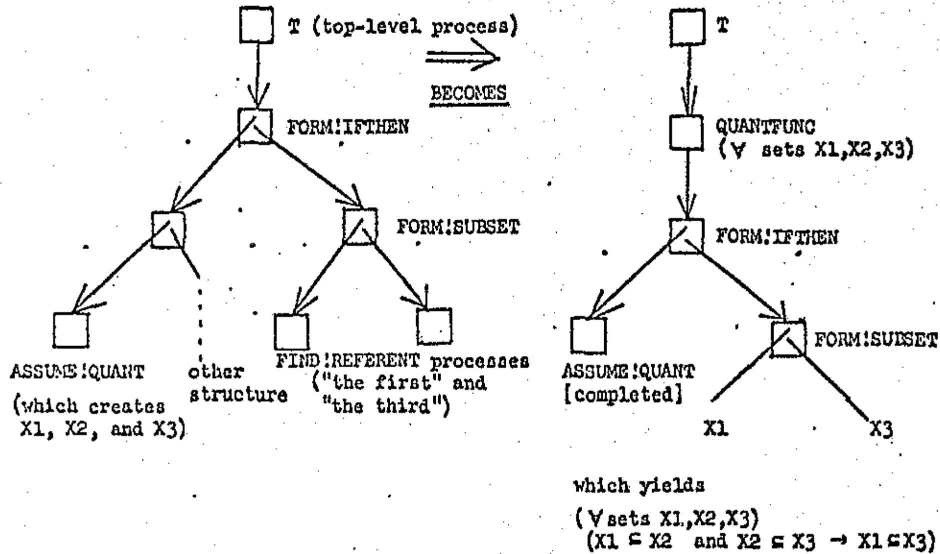

Figure 10. Dependency Graphs for (11)

4.4 Semantic Functions with Side Effects

In this section and the next we describe some of the other problems that PLISP should assist us in handling and indicate how we would approach them. The first example concerns a sentence that might occur in an informal mathematical argument; understanding this sentence and converting

it to first-order form for our proof-checking system require handling context and side effects.

The sentence

(12) Take the next set in the sequence and form its powerset.

as shown in Figure 11 presents an interesting problem: the process associated with the node labelled NEXT accesses contextual memory to determine what the next set is, and has the side effect of incrementing an index in that memory. The process resolving the pronominal reference of the word "its" in the node labelled ITS must find the value returned by the process associated with the phrase "the next set in the sequence". Copying the syntactic subtree associated with the antecedent is not only inefficient, but when side effects are involved, it may well be wrong, just as (12) means something different than:

(12') Take the next set in the sequence
and form the powerset of the next set in the sequence.

One might claim that pronouns are not only used for reasons of syntactic style, but also to avoid unwanted side effects.

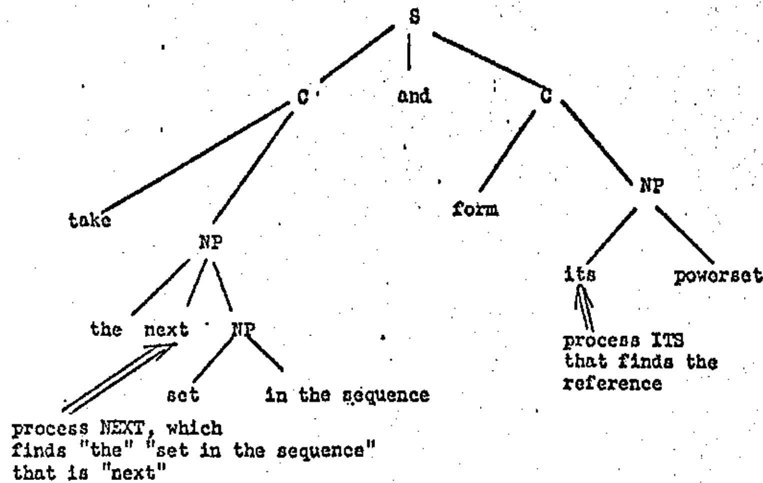

Figure 11. Syntax Tree for Evaluation with Side Effects

While this sentence itself has no convenient first-order representation, within the context of a mathematical argument wherein the sequence is given, a first-order representation may be obtainable. Solving the problem of understanding a mathematical proof so that a proof checking system can check that proof remains a difficult problem.

4.5 Other Operators

Certain words, such as "respectively" and "same", create global changes in the evaluation even though they occur quite locally within the original syntax. For example,

(13) Two sets are equal just in case they have the same elements.

The surface syntax tree for (13) is given in Figure 12.

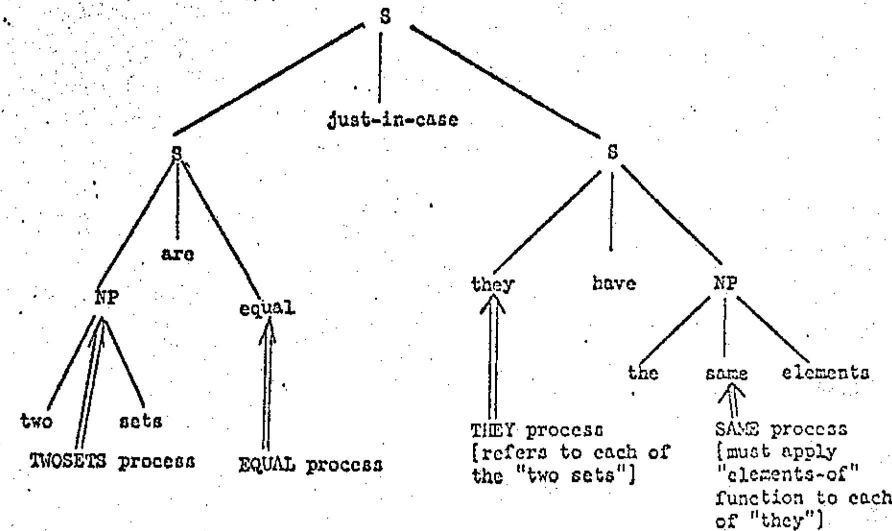

Figure 12. Syntax Tree Showing the Word "Same"

The problem processes are:

1. The process labelled TWOSETS must create two variables, say X1 and X2.

2. The process EQUAL must restructure that subtree of the sentence to represent the meaning that the "two sets", now represented by X1 and X2, are equal. This syntax typically occurs only with symmetric relations, as in "the two women are sisters"; "the two numbers are greater" will prompt the hearer to ask "Than what?"

3. "They" must denote X1 and X2, and the process labelled THEY must determine this. The result is to create a new quantifying process above the top node IFF, and a universal quantification over X1 and X2.

4. The word "same" occurs within the context "they [i.e., X1 and X2] have the same elements". "Have", in mathematics, means function application. The process associated with "same" applies the function to each of X1 and X2 and asserts that the elements of X1 are all elements of X2 and conversely, i.e., we must end up with something like:

$$(\forall X3)(X3 \in X1 \Leftrightarrow X3 \in X2)$$

5 Comparisons with Previous Work

5.1 Logical Theories of Natural Language

Few of our specific insights about scopes and operators are particularly new. For example, logicians have explicated the "logical forms" of many similar sentences without suggesting what computational mechanisms are appropriate for determining them. Indeed, the major mechanism assumed by logicians such as Montague [6] and Suppes [9] corresponds to simple endorder tree traversal as in LISP EVAL; we regard this to be too simple. The model presented here provides more powerful computational mechanisms for processing natural language.

A classical philosophical claim about language, the Frege-Davidson hypothesis, can be interpreted as suggesting that endorder tree traversal is conceptually sufficient to evaluate meaning. While this claim is strictly correct in the sense that any effective semantic evaluation mechanism can be encoded into an endorder tree traversal, the resulting semantic functions are overly complex and counterintuitive. We have attempted in this paper to show some examples of sentences that are not

readily analyzed by semantic functions using endorder tree traversal as the control structure.

5.2 Transformational Grammar

The goal of transformational grammar has been to generate from the surface syntax a syntactic "deep structure" that also represents the semantics of the sentence in a way that satisfies the Frege-Davidson hypothesis. Once this deep structure has been computed, the semantics of the sentence can be evaluated in the manner suggested by the logicians.

There are several computational and conceptual advantages to PLISP. The transformational approach maintains that the determination of control structure can be made solely on the basis of the syntax of the surface-level sentence, plus the transformational rules. This is awkward because many of the factors involved in this determination are themselves "semantic" or "pragmatic". For example, the information needed in our examples to determine operator scope and pronominal reference is of this nature. The "deep structures" of the PLISP evaluation--the dependency graphs that arise as a side effect of the evaluation process--do not have to be trees. This is sometimes significant, as in example (10) above.

5.3 Winograd's Work

The key value of the work of Winograd [11] is his demonstration that one must have various kinds of information available at all points in processing natural language in order to get reasonable results. One may disagree with any of the details of his SHRDLU system and still be forced

to recognize the validity of his insights about what information is needed at what point of the understanding process.

The result of this is, however, an increase in the complexity of the resulting programs. The relatively neat modules representing "syntax", "semantics" and so on have been replaced by large "systems" which, while more accurately representing the required flow of information, nevertheless become unwieldy as programs. One solution is to introduce new programming systems that will allow one to modularly express the more complex relationships.[7]

6 Conclusion

The intention of PLISP is to express modularly in a process the special information about the local and global effects that a given syntactic construction can have. In particular, as illustrated in our examples, we have attempted to avoid cluttering functions with subroutines that decode the arguments to the main function and then attempt to do some of the computation that should have been done in other functions.

The remaining problems concern an actual implementation of PLISP. We intend to use the "spaghetti stack" features of INTERLISP-10, which provide multiple contexts, as the basis of the implementation. Debugging these functions is likely to be difficult; the "deadlock" and "race" problems familiar in operating systems theory will arise here. Yet we feel that it is important to learn how to use multi-process control structures in the context of natural language understanding.

[7] Some of Winograd's recent work [12] has taken this approach.

References

1. Bobrow, D. G. and Wegbreit, B. A model and stack implementation of multiple environments. Communications of the ACM, 1973, 16, 591-603.

2. Bobrow, D. G., Burchfiel, J. D., Murphy, D. L., and Tomlinson, R. S. TENEX: A paged time sharing system for the PDP-10. Communications of the ACM, 1972, 15, 135-143.

3. Knuth, D. E. Semantics of context free languages. Mathematical Systems Theory, 1968, 2, 127-145.

4. Lampson, B. W., Mitchell, J. G., and Satterwaithe, E. H. On the transfer of control between contexts. Programming Symposium 1974. Springer Verlag, Springer Lecture Notes in Computer Science 19, New York 1974.

5. McDermott, D. V., and Sussman, G. J. From PLANNER to CONNIVER: A genetic approach. AFIPS Conference Proceedings, 1972, 41, 1171-1179.

6. Montague, R. English as a formal language. Formal Philosophy: Selected Papers of Richard Montague, edited by R. H. Thomason, Yale University Press, New Haven, 1974.

7. Rulifson, J. F., Derksen, J. A., and Waldinger, R. J. QA4: a procedural calculus for intuitive reasoning. SRI AI Technical Note 73, 1972.

8. Smith, R.L., and Blaine, L. H. A generalized system for university mathematics instruction. Proceedings of an ACM SIGCSE-SIGCUE Joint Symposium, Anaheim, California, February 12-13, 1976, 280-288.

9. Suppes, P. C. Semantics of context free languages (Tech. Rep. No. 171). Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1971.

10. Teitelman, W. INTERLISP Reference Manual. Xerox Corporation, Palo Alto, California (1975).

11. Winograd, T. Procedures as a representation for data in a computer program for understanding natural language. MIT AI TR-17 (MIT Project MAC TR-84), 1971.

12. Winograd, T. Breaking the complexity barrier again. SIGPLAN Notices, 1975, 10, 13-30.