18

"1965 International Conference on Computational Linguistics"

A Heuristic Approach to Natural Language Processing

Denis M. Manelski and Gilbert K. Krulee

Shell Oil Company
50 West 50th Street
New York 20, New York

Northwestern University
Technological Institute
Evanston, Illinois

ABSTRACT

This paper is concerned with the design of a processor capable of formalizing English language descriptions of problems in the sentential calculus. The emphasis is on the design of a system with natural language processing capabilities, but the formal languages specified are oriented to the problem context.

A series of automata are specified to carry out the necessary functions. The automata identify the premises in the problem strings, specify the appropriate logical connectives among the premises and determine which premises are meaning-equivalent. The syntax of each automaton is defined and examples are used to illustrate their functioning.

The automata accept statements in the language L1, the set of English statements of problems in the sentential calculus. The individual premises $p \in$ L1 are recognized by the syntax Γ, where Γ is chosen so that the language L2 recognized by it is a subset of L1. Furthermore, the strings in L2 are restricted to the declarative sentences. Once the premises and their logical connectives have been identified, those that are meaning-equivalent are located in two additional steps. First the L2 description of the string is mapped into a string in L3. The L3 language consists of a limited set of canonical forms that ease the problem of establishing meaning equivalence of premises. Finally, the automaton applies heuristically a sequence of problem-oriented and meaning-preserving transformations in order to establish meaning-equivalence. Two premises are taken to be meaning-equivalent if one can be deduced from the other. Otherwise, they are taken to be not meaning-equivalent.

A HEURISTIC APPROACH TO NATURAL LANGUAGE PROCESSING[1]

Introduction

The recent evolution of programming languages has tended to improve communication between man and computer. The use of mnemonics, automatic storage allocation, English-like operators (such as in COBOL) and problem-oriented languages has greatly facilitated the task of the programmer. Thus, the solution algorithm for a large class of computational problems can be defined with relative ease in languages such as FORTRAN and ALGOL, specifically designed for these classes of problems.

This paper describes an attempt to further simplify the communication between programmer and computer by defining a system which can produce a formal description from its natural (verbal) input.[2]

In order to study this approach a specific problem area was chosen, the propositional or statement calculus. It will be evident that the problem area chosen has influenced the design of the system; nonetheless it should be clear that the linguistic capabilities of the system are general rather than specific to the problem context.

In designing this processor, two major abilities are required. First, the processor must be able to identify each elementary premise and all logical connectives. It must also determine which premises are to be taken as equivalent.

[1] This research was supported by Grant G-17951 of the National Science Foundation. A majority of the system has been programmed in the list processing language IPL-V (Newell, 1961).

[2] For a more complete description and some program listings see Manelski, 1964.

The processor is composed of three series coupled automata (see Fig. 1). The first automaton, A1, accepts as its inputs the language L1, where L1 is the set of all English language statements of problems in the propositional calculus. This automaton is concerned with the identification of the premises and logical connectives of a problem. This is achieved by using a syntax Γ capable of recognizing strings in L2, where L2 is a subset of L1. The syntax Γ consists of a hierarchy of syntaxes; a phrase structure syntax $\Gamma 1$ designed to recognize a subset of English composed of simple declarative sentences and the set of transformations specified by ΓT.[1]

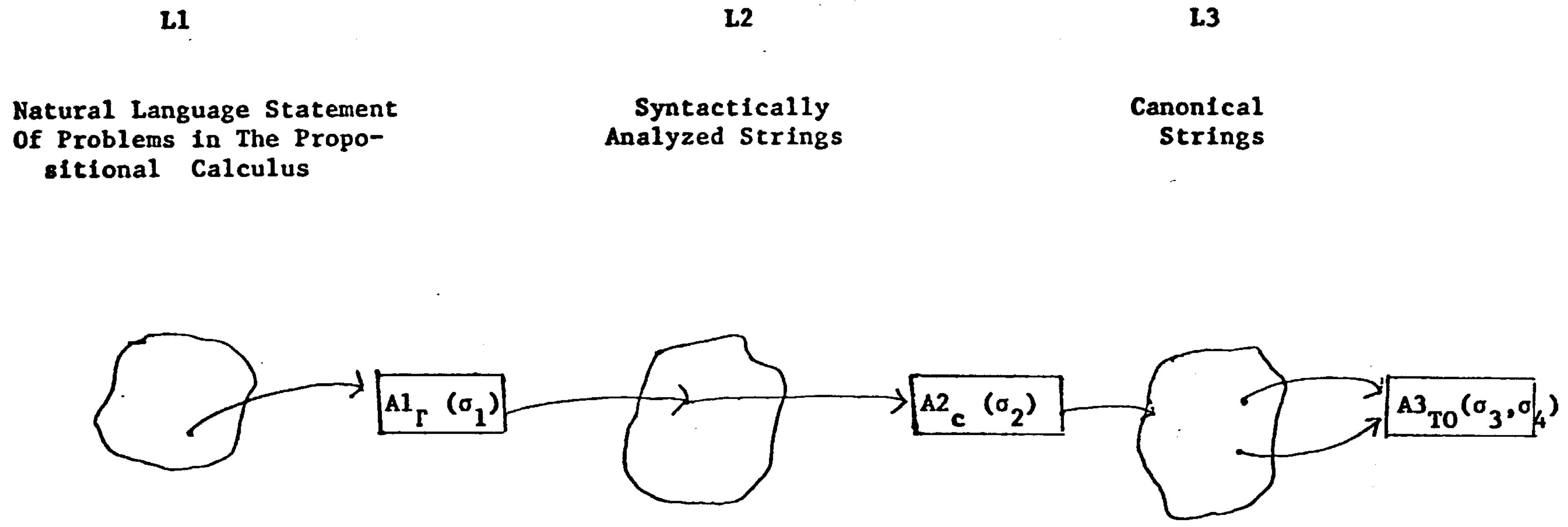
The equivalent premises are identified by the automata A2 and A3. The automaton A2 maps a premise, identified by A1, into a canonical form specified by the syntax C that defines the language L3. This step is designed to facilitate the distinction of equivalent premises. Finally, A3 applies a sequence of meaning preserving transformations from the set TO = $\{T_1, T_2, \ldots, T_m\}$ on the string $\sigma_r, \sigma_s \in L_3$ such that if:

$$T_i T_j \ldots T_\ell (\sigma_r) = \sigma_s \qquad \text{with } T_k \in TO$$

the two strings are considered meaning equivalent. Should the system be unable to find a deduction satisfying these conditions or under certain other heuristically chosen criteria the strings are assumed to represent different premises.

In order to test the system described in this paper, problems were drawn from Stoll (1961). Some will be used later to illustrate the capabilities and inadequacies of the present system.

[1] Chomsky's discussion of transformations and the inadequacies of various models for natural languages can be found in the monograph "Syntactic Structures".


L1
Natural Language Statement Of Problems in The Propositional Calculus
L2
Syntactically Analyzed Strings
L3
Canonical Strings
$A1_{\Gamma}\ (\sigma_1)$
$A2_{c}\ (\sigma_2)$
$A3_{TO}(\sigma_3,\sigma_4)$

A Schematic Diagram Of The Series Coupled Automata

<u>Figure 1</u>

Each of the automata will be discussed in two ways, first in terms of its syntax. Finally the information flow for its implementation as a computer program will be outlined.

Characteristics of the Natural Language Processor (A1)

The automaton A1, as mentioned in the previous section, consists of two completely different syntactic mechanisms. The system includes a phrase structure syntax designed to recognize an extremely restricted subset of the English language, simple declarative sentences. The syntax of the processor also includes a limited set of transformations chosen to enhance the power of the language generated, but also specifically chosen for the problem context.

If we consider the syntax of A1, Γ, as consisting of $\Gamma 1$ and ΓT we have defined a hierarchy of languages:

$$L1 \supset L2 \supset L\Gamma 1$$

Here L1 consists of all the legal problem statements; L2 consists of the set of strings recognized by Γ; and $L\Gamma 1$ consists of all the strings recognized by the syntax $\Gamma 1$. Thus, the syntax Γ of the automaton A1 is really composed of two disjoint sets of rewriting rules, $\Gamma 1$ and ΓT. The syntax $\Gamma 1$ is a phrase structure grammar designed to generate or recognize a subset of English composed of simple declarative sentences. The syntax ΓT contains a set of transformations designed for the purposes of isolating premises and specifying logical connectives. This hierarchy can be visualized in Figure 2.

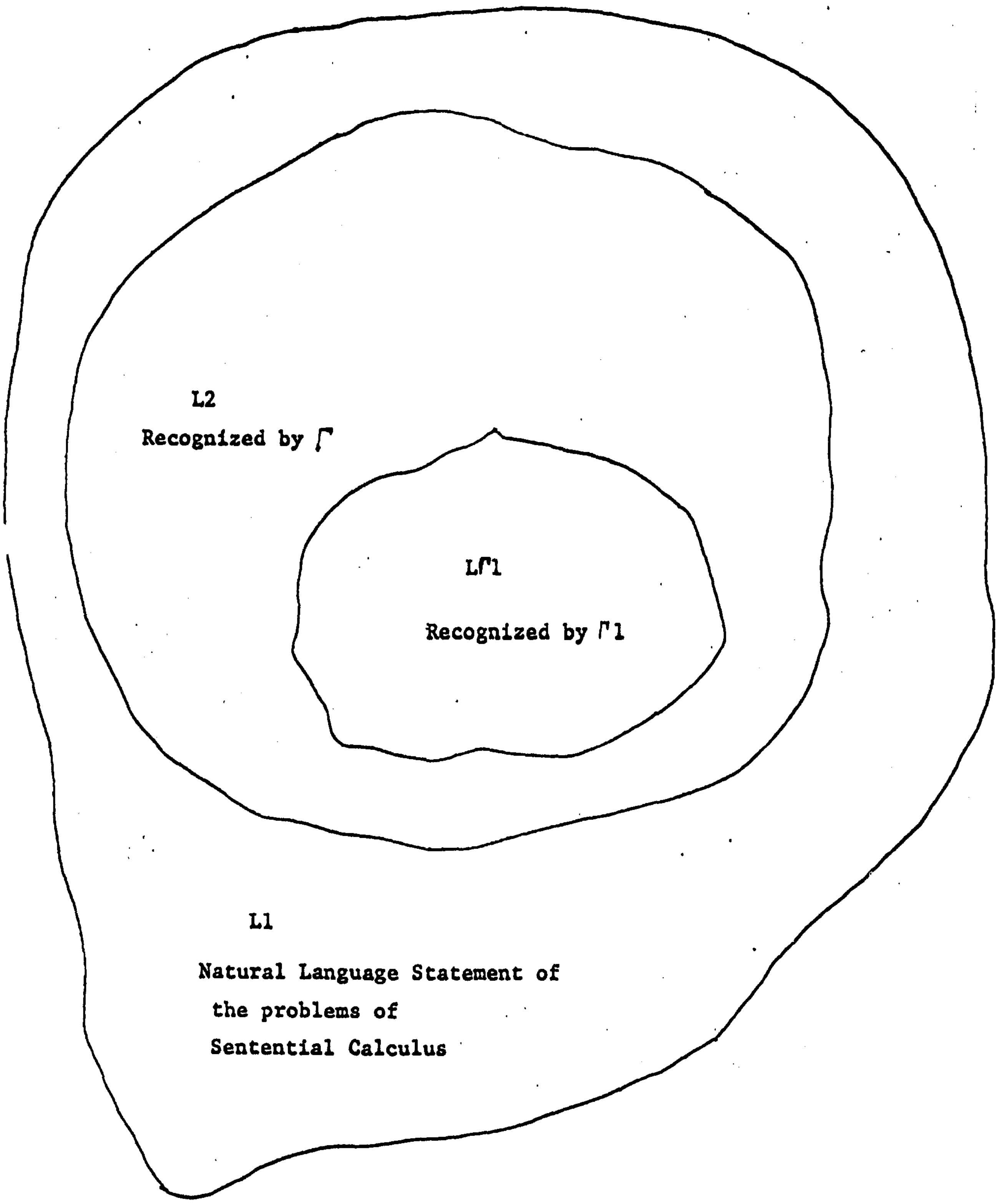

L2
Recognized by Γ
$L\Gamma 1$
Recognized by $\Gamma 1$
L1
Natural Language Statement of the problems of Sentential Calculus

Figure 2

Initially, we shall describe the class of sentences recognized by $\Gamma 1$, and then characterize the strings recognized by Γ. From the following discussion it will be made clear that we are building a recognizer rather than a generator. The automaton A1 will not perform syntactic analysis below the level of the alphabet (i.e., words) of the language. Thus, the processor would recognize:

The bridge was high

The bridges was high

as the same sentence since the differences are at a level below that specified by its syntax.

The processor consists of an alphabet A, where:

$$A = N \cup D \cup PN \cup ADJ \cup VEQ \cup VTR \cup VIN \cup VFAC \cup VAUX \cup PREP \cup ADV \cup THAN \cup ADJC$$

with the sets representing:

N:	noun
D:	determiner
PN:	pronoun
ADJ:	adjective
VEQ:	verb equational
VTR:	verb transitive
VIN:	verb intransitive
VFAC:	verb factitive
VAUX:	verb auxiliary
PREP:	preposition
ADV:	adverb
ADJC:	comparative adjective
THAN:	Than

Although the task of the assignment of word classes is that of the linguist, in general, if X_i and X_j are sets comprising A we expect

$$X_i \cap X_j \neq \phi \qquad \text{for } i \neq j$$

where ϕ represents the empty set. The occurrence of an element of the alphabet in more than one word class is known as homography and is common to the natural languages.

For purposes of derivation, we distinguish between the elements of the alphabet, to be known as the "terminal" elements, and the symbols from the syntax such as S, NP, ADJ, etc., which will be referred to as the nonterminals. The word assignments might be as shown in Table 1.

N	=	man, boy, house,...
D	=	a, the,...
PN	=	he, they,...
ADJ	=	blue, large,...
VMEQ	=	is, are,...
VMTR	=	hit, hits,...
VMINTR	=	rained, went,...
VMFAC	=	appoint, call,...
VAUX	=	will, should,...
PRP	=	in, to,...
ADV	=	quickly, slowly,...
ADJC	=	larger, better,...

Table 1

Although the processor is limited in the size of the available dictionary, for purposes of discussion no limitations will be assumed.

In addition it is necessary to specify the syntax of the recognizer, which uses the rewriting rules of the axiomatic system $\Gamma 1$ in Table 2. Examining the syntax $\Gamma 1$, we see that it meets all the requirements of a phrase structure grammar. Also, $\Gamma 1$ generates several classes of strings characterized by the verb type. Since this classification will be fundamental to the design of A2, we shall give some examples in L2 and later show the mapping of A2.

Syntax #1 for Processor A1

1)	S → NP + VP	sentence, noun phrase, verb phrase
2)	NP → D + N	determiner, noun
	N	
	PN	pronoun
3)	NP → NP + PRP	prepositional phrase
4)	PRP → PREP + NP	preposition
5)	N → ADJ + N	adjective
6)	ADJ → ADJ + ADJ	
7)	PADJC → ADJC + than + NP	comparative adjectival phrase, comparative adjective
8)	NP → NP + NP	
9)	VP → VEQ + PREDEQ	verb equational, equational predicate
	VTR + PREDTR	verb transitive, transitive predicate
	VITR + PADV	verb intransitive, adverbial phrase
	VFAC + PREDFAC	verb factitive, factitive predicate
10)	VEQ → VMEQ	main verb, equational
11)	VMEQ → VAUX + VMEQ	verb auxiliary
12)	PREDEQ → NP	
	ADJ	
	PRP	
	PADJC	
13)	VITR → VMITR	main verb, intransitive
14)	VMITR → VAUX + VMITR	
15)	PADV → ∅	empty
	ADV	
	N	
	PRP	
16)	PADV → PADV + PADV	
17)	VTR → VMTR	main verb, transitive
18)	VMTR → VAUX + VMTR	
19)	PREDTR → NP	
	NP + PADV	
	NP + PRP	
	NP + NP	
20)	VFAC → VMFAC	main verb, factitive
21)	VMFAC → VAUX + VMFAC	
22)	PREDFAC → NP	
	NP + NP	
	NP + PADV	

Table 2

The syntax Γ1 identifies four verb types, equational verbs, intransitive verbs, transitive verbs, and factitive verbs with their corresponding predicates. The following examples show some of the possible sentences:

Equational verb:

(i) John is home.

(ii) John is tall.

(iii) John is by the house.

(iv) John is taller than Peter.

A derivation of (ii) in the syntax Γ1 is

(S(NP(N John))(VP(VEQ(VMEQ is))(PREDEQ(ADJ tall))))

Intransitive verb:

(i) The Dodgers win.

(ii) The Dodgers win seldom.

(iii) The Dodgers win money.

(iv) The Dodgers win at home.

The derivation of (i) is

(S(NP(D The)(N Dodgers))(VP(VITR win)(PADV ∅)))

Transitive verb:

(i) John loves Mary.

(ii) John loves the winnings from the track.

The derivation for (i) is

(S(NP(N John))(VP(VTR(VMTR loves))(PREDTR(NP(N Mary)))))

Factitive verb:

(i) John called home.

(ii) John called his friend a fool.

The derivation of (i) is

(S(NP(N John))(VP(VFAC(VMFAC called))(PREDFAC(NP(N Mary)))))

Several types of sentences will not be recognized by Γ1. Some of these could be included by additional productions. Some additional types of sentences will be recognized when ΓT is added to the syntax. Other sentence forms are not considered necessary within the original problem context. Let us list some of the sentences in L1 that are outside of the capabilities of recognition with Γ1.

Imperative sentences:

Go home.

Interrogative sentences:

Is John coming home?

Passive sentences:

Home is where John should be.

Conditional sentences:

If John should come home...

Compound sentences:

John will go home and Mary will stay.

Complex sentences:

John, should he so desire, will go home.

In order to make the processor A1 useful in the problem context, it is necessary to increase the class of strings in L2. In contrast to the syntax Γ1, which uses the rewriting rules on the non-terminals in the deduction string, the transformation set ΓT is designed to operate on the derivations in Γ1. Generally, transformations have been discussed in terms of generators. Attention has been focused on increasing the class of strings that a formal language can generate (39). However, our problem is to use ΓT in order to simplify the class of strings that $\Gamma 1$ will have to recognize. Thus, our transformation set ΓT should decompose the string

John will go home and Mary will stay.

into the following simpler strings:

(i) John will go home.

(ii) Mary will stay.

Since we are interested in formalizing the natural language inputs as statements in the sentential calculus, the transformations will also give us information as to the appropriate logical connectives for the premise. Thus, in the previous example our processor could be expected to define a statement of the form:

$$p \wedge g$$

In order to explore the powerful linguistic possibilities of transformations, a limited number were chosen. We shall now define the transformations and show how the linguistic capabilities of A1 have been increased.

The transformation set ΓT presently contains as its axioms:

$$T = \{TNOT,\ TCOM,\ TCOND\}$$

In order to specify a transformation, we must not only define the structural changes it produces but also the class of strings to which it is applicable. The transformations, as defined in ΓT were adapted for A1. Since we are not interested in generating grammatically correct English sentences, but rather mapping the input strings into a form recognizable to $\Gamma 1$, it is possible to omit the transformations for tenses because they operate at a level lower than that of the terminals. By implication $\Gamma 1$ will process strings that are not grammatically correct. Thus, if A1 were presented with the sentence:

If it were cold tomorrow,....

the transformation TCOND will give as its output:

It were cold tomorrow.

This premise would still be processed although it is grammatically incorrect.

Another difference between the transformations as specified by Chomsky, and those used by A1 is in the direction of the mapping. The ΓT transformations have L2 as their domain and the kernel strings generated by $\Gamma 1$ as their range. This is the inverse of the mappings considered by Chomsky (1957).

TNOT: is defined on strings of the form

(i)..+NP+VAUX+not+VMTR+...

(ii)..+NP+VAUX+never+VMTR+...

(iii)..+NP+VMEQ+not+...

(iv)..+NP+VMEQ+never+...

(v)..+NP+VAUX+not+VMEQ+...

(vi)..+NP+VAUX+never+VMEQ+...

(vii)..+NP+VAUX+not+VITR+...

(viii)..+NP+VAUX+never+VITR+...

(ix)..+NP+never+VITR+...

(x)..+NP+VAUX+never+VTR+...

(xi)..+NP+VAUX+never+VTR...

(xii)..+NP+never+VTR+...

(xiii)..+NP+VAUX+not+VFAC+...

(xiv)..+NP+VAUX+never+VFAC+...

(xv)..+NP+never+VFAC+...

Should a string σ_1 correspond to one of the above patterns TNOT(σ_1) becomes:

(i)..+NP+VAUX+VMTR+...

(ii)..+NP+VAUX+VMTR+...

(iii)..+NP+VMEQ+...

(iv)..+NP+VMEQ+...

(v)..+NP+VAUX+VMEQ+...

(vi)..+NP+VAUX+VMEQ+...

(vii)..+NP+VAUX+VITR+...

(viii)..+NP+VAUX+VITR+...

(ix)..+NP+VITR+...

(x)..+NP+VAUX+VTR+...

(xi)..+NP+VAUX+VTR+...

(xii)..+NP+VTR+...

(xiii)..+NP+VAUX+VFAC+...

(xiv)..+NP+VAUX+VFAC+...

(xv)..+NP+VFAC+...

Examples of some of the cases follow:

σ_1: John will never hit Mary.

TNOT(σ_1): John will hit Mary.

σ_2: Today is not cold.

TNOT(σ_2): Today is cold.

σ_3: Tomorrow will not be cold.

TNOT(σ_3): Tomorrow will be cold.

σ_4: John never suffers.

TNOT(σ_4): John suffers.

TCOM: operates on strings in the following domain only:

(i) ..+S_1+and+S_2+ ...

(ii) ..+S_1+,+S_2+ ...

(iii) ..+S_1+or+S_2+ ...

(iv) ..+S_1+then+S_2+ ...

(vi) Either +S_1+or+S_2+ ...

(vii) Therefore+,+Either+S_1+or+S_2+ ..

The range of the function is any string with the following format:

S_1

S_2

Here the information between "S_1" and "S_2" is used by the processor only to establish the Boolean connectives for the statements. Some examples will show the effect of TCOM on strings σ in the domain of the transformation.

σ_1: Either Sally and Bob are the same age or Sally is older than Bob.

TCOM(σ_1): Sally and Bob are the same age.

Sally is older than Bob.

σ_2: The races are fixed or the gambling houses are crooked.

TCOM(σ_2): The races are fixed.

The gambling houses are crooked.

TCOND: is defined over strings with the following configuration:

(i) ..+If+S_1+...+,then+S_2+....

(ii) ..+If+S_1+...+,+S_2+....

and has as its range the following forms:

..+S_1+...

..+S_2+...

As in the other transformations its application defines the logical connectives for A1.

We can see the effect of TCOND on the following strings:

σ_1: If the Dodgers win, then Los Angeles will celebrate.

TCOND(σ_1): The Dodgers win.

Los Angeles will celebrate.

The definitions of the syntactic elements used in establishing the domain of ΓT are given by the phrase-structure grammar Γ1. Another convention used in the discussion is to allow a series of dots (....) to refer to any syntactic structure. It is also implied that the transformations may be concatenated as necessary.

To illustrate their use, we utilize the following examples:

σ_1: If the Dodgers win, then Los Angeles will celebrate, and if the White Sox win, Chicago will celebrate.

TCOND(σ_1): The Dodgers win.

Los Angeles will celebrate and if the White Sox win, Chicago will celebrate.

$TCOM(TCOND(\sigma_1))$: The Dodgers will win.

Los Angeles will celebrate.

If the White Sox win, Chicago will celebrate.

$TCOND(TCOM(TCOND(\sigma_1)))$: The Dodgers will win.

Los Angeles will celebrate.

The White Sox win.

Chicago will celebrate.

σ_2: If I miss my appointment and start to feel downcast, then I should not go home.

$TCOND(\sigma_2)$: I miss my appointment and start to feel downcast.

I should not go home.

$TCOM(TCOND(\sigma_2)$: I miss my appointment.

Start to feel downcast.

I should not go home.

$TNOT(TCOM(TCOND(\sigma_2))$: I miss my appointment.

Start to feel downcast.

I should go home.

In this example the resultant strings are not recognizable by $\Gamma 1$. Thus, "start to feel downcast" has its subject implied by the preceding string, and could be thought of as "I start to feel downcast". Some of the difficulties caused by the transformations can be overcome by A1.

Description of the Natural Language Processor (A1)

In order to design a processor of the type described in the previous section it is necessary to specify the relationship between the recognition rules $\Gamma 1$ of the phrase structure grammar and the rewriting rules ΓT of the set of transformations. Clearly $\Gamma 1$ and ΓT are interdependent since the input cannot always be analyzed in terms of the syntax $\Gamma 1$ and because the rewriting rules of ΓT are defined in terms of $\Gamma 1$. Perhaps an example illustrates this point more effectively. Consider the input string:

If John went to the store then Mary went home.

This is clearly a case in which we should apply TCOND $\in$ T in order to obtain:

S1 - John went to the store.

S2 - Mary went home.

However, the processor cannot find S1 and S2 because they are defined in terms of $\Gamma 1$ which cannot determine S1 and S2 since it cannot analyze strings such as "If John went to the store...". This vicious circle has been resolved by determining heuristically when the transformations should be applied. If the strings resulting from the application of the transformations cannot be analyzed by $\Gamma 1$, the system attempts to apply the transformations again.

The general hierarchy of the programs can be found in Figure 3- . The program DO embodies the essential features of the automaton A1. A brief description of the various sub-routines involved will serve to illustrate the workings of the processor and the difficulties that it might encounter.

The automaton A1 can be considered as having two quite distinct functions. Initially, certain key words are marked in the problem input (giving rise to the hypothesized input string) and later the set of transformations are used in conjunction with the marked words to generate possible premises (to be called "input strings").

The necessary information can be more fully explained by considering a program D0 designed to implement A1 (see Figure 3). The program D0 initially calls the sub-routine D15 which performs a left-to-right scan on the problem string. All elements of the set MTO (where

MTO = {if, then, and, or, not, never, either, therefore, then, ,}

the last two elements are the symbols ", then" and ",") are marked. After marking, the problem string becomes both the input string (i.s.) and the hypothesized input string (h.i.s.). The syntactic analysis of an h.i.s. is attempted by E0. Failing to find a satisfactory parsing, control is transferred by D2 to D1; otherwise control goes to D13. The sub-routine D13 searches for an additional h.i.s.; on finding one, it deletes the successfully parsed string from the i.s. and the list of h.i.s. Should no other h.i.s. be found, the executive calls D14 which halts the program. After performing the necessary output functions, D1 scans the h.i.s. currently being processed. If any marked words are found, control is passed to D3; otherwise the transfer is to D11. D11 erases the previous h.i.s. and replaces them (i.e., all of them) with the i.s. Should D1 find that some of the words are marked, the processor

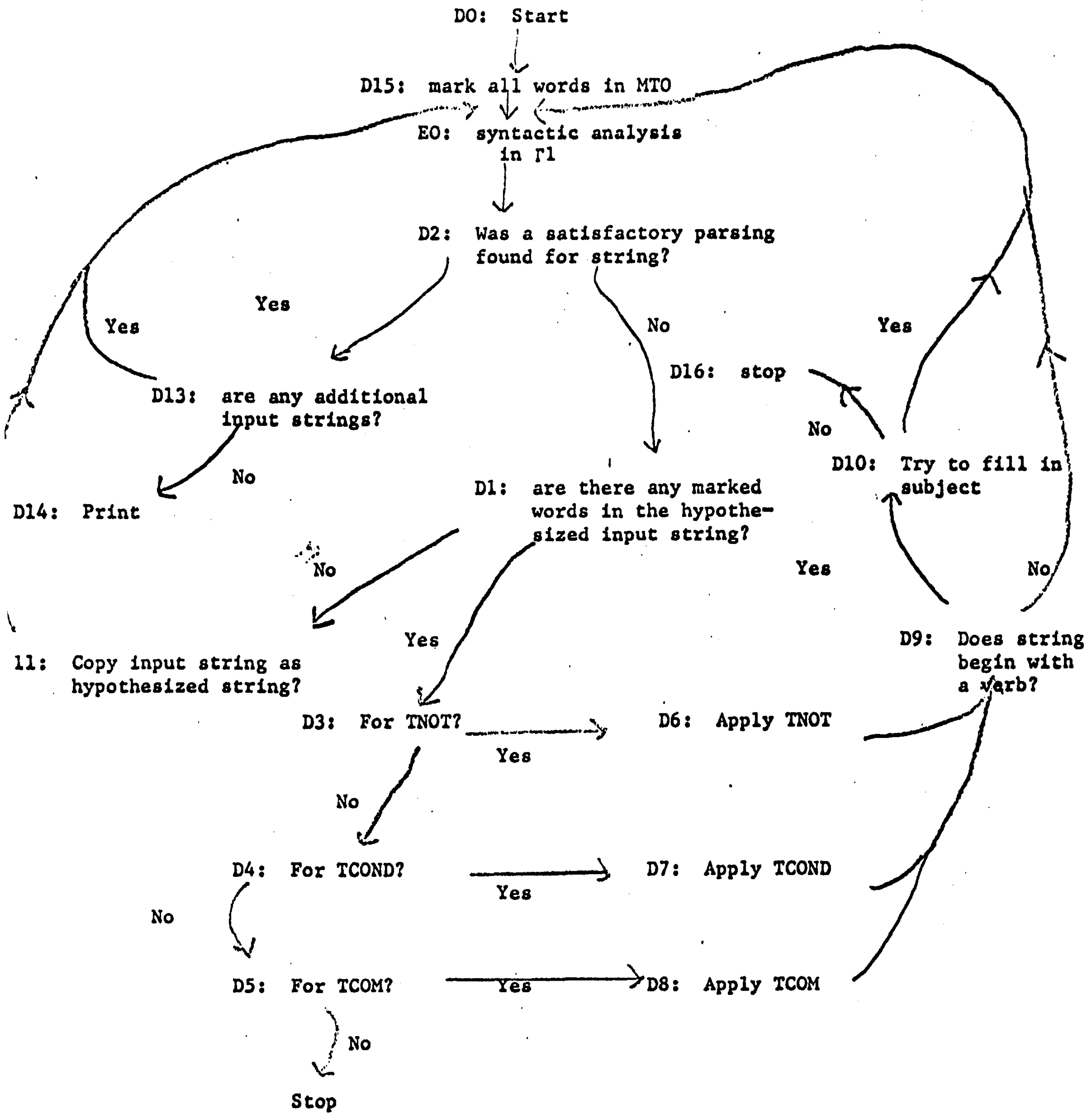

D0: Start
D15: mark all words in MTO
E0: syntactic analysis in Γ1
D2: Was a satisfactory parsing found for string?
Yes
No
Yes
Yes
D16: stop
D13: are any additional input strings?
No
No
D10: Try to fill in subject
D14: Print
D1: are there any marked words in the hypothe-sized input string?
No
Yes
No
Yes
D9: Does string begin with a verb?
11: Copy input string as hypothesized string?
D3: For TNOT?
Yes
D6: Apply TNOT
No
D4: For TCOND?
Yes
D7: Apply TCOND
No
D5: For TCOM?
Yes
D8: Apply TCOM
No
Stop

Figure 3

attempts to apply the transformations TNOT, TCOND, or TCOM by using the test routines D3, D4 or D5 in transferring control to D6, D7 or D8, respectively. D3 transfers control to D6 when "<u>not</u>" or "<u>never</u>" (the underlining is used in this section to indicate the symbols as marked) are in the h.i.s.; D6 deletes the marked symbol from the h.i.s. The sub-routine D5 is only applied when the h.i.s. begins with "<u>if</u>"; it in turn transfers control to D7 which deletes the first of the marked symbols "<u>then</u>", "<u>therefore</u>", "<u>,</u>" or "<u>, then</u>" that it finds in the h.i.s. While removing the marking from the corresponding symbol in the i.s. two new h.i.s. are created by dividing the list at the location of the marked symbol. D5 and D8 are similar to D4 and D7; however, division of the h.i.s. is done on "<u>and</u>", "<u>or</u>" or with the symbol "<u>either</u>" being erased from the beginning of the h.i.s. if it is present. The routines D6, D7 and D8 transfer control to D9 which is called to test whether the h.i.s., being processed, begins with a verb: if this condition exists HO attempts to precede it with the first noun or pronoun of the previous h.i.s. Should it not be possible for the processor to carry out this operation, the program prints out the syntactic analysis it has accomplished and halts. Both D10 and D9 transfer to E0.

Some examples will clarify the logic of D0. Let the input string σ_1 be:

σ_1: John and Mary went home.

The branching of the problem would be

D0: transfers control to D15.

D15: marks the word "and"; the h.i.s. is "John <u>and</u> Mary went home (the underlining indicates the marked word).

E0: parses "John and Mary" went home.

D13: there are no additional h.i.s.

D14: stop.

σ_2: John went home and Mary went to the store.

D0: transfers to D15.

D15: the i.s. and h.i.s. become John went home <u>and</u> Mary went to the store.

E0: fails to parse the sentence.

D2: transfers to D1.

D1: transfers control to D3.

D3: control parses to D4.

D4: transfers control to D5.

D5: transfers control to D8.

D8: the i.s. becomes

John went home and Mary went to the store.

while the h.i.s. become

John went home.

Mary went to the store.

D9: after testing the h.i.s. at the top of the pushdown list (John went home) transfers control to E0.

E0: successfully parses the current h.i.s.

D2: transfers control to D13.

D13: locates the next h.i.s.

E0: successfully parses the h.i.s. at the top of the pushdown list (Mary went to the store).

D2: transfers the processor to D13.

D13: cannot locate any additional h.i.s.

D14: prints the results of the parsing.

σ_3: If John, Peter and Paul were at the game,...

D0: calls D15.

D15: marks the problem string as "$\underline{\text{If}}$ John$\underline{,}$ Peter $\underline{\text{and}}$ Paul were at the game$\underline{,}$,...." which is copied as the h.i.s.

E0: fails to find a deduction for the h.i.s.

D2: transfers control to D1.

D1: transfers control to D3.

D3: transfers control to D4.

D4: transfers control to D7.

D7: the marked words have the structure required for TCOND and changes the i.s. to

"$\underline{\text{If}}$ John, Peter $\underline{\text{and}}$ Paul were at the game$\underline{,}$...."

and the h.i.s. become

"John"

"Peter $\underline{\text{and}}$ Paul were at the game$\underline{,}$..."

D9: the h.i.s. does not begin with a verb.

E0: fails to find a parsing.

D2: transfers control to D1.

D1: the h.i.s. "John" has no marked words.

D11: the previous i.s. becomes the h.i.s.

"If John, Peter and Paul were at the game,..."

E0: fails to find a parsing.

D2: transfers control to D3.

D3: calls sub-routine D4.

D4: finds the marked "If" and "," calling for TCOND.

D7: the h.i.s. become

"John, Peter"

"Paul were at the game,..."

and the i.s. is marked as

"If John, Peter and Paul were at the game,..."

D9: the h.i.s. does not begin with a verb.

E0: a satisfactory parsing cannot be found.

D2: transfers the processor to D1.

D1: there are no marked words in the h.i.s.

D11: the h.i.s. becomes

"If John, Peter and Paul were at the game,..."

E0: fails to find a parsing.

D2: transfers control to D3.

D3: calls D4.

D4: finds the "If" and "," for TCOND.

D7: the new h.i.s. is formed

"John, Peter and Paul were at the game" (the remainder of the sentence is a separate h.i.s.).

the i.s. is changed to

"_If_ John, Peter and Paul were at the game,..."

D9: transfers the processor to E0.

E0: analyzes the first h.i.s. The program would then analyze the remainder of the sentence.

As indicated in the above examples the parsing of the i.s. is attempted by sub-routine E0, using the syntax specified in Table 2. The presently implemented version of E0 uses a bottom-to-top search in the sense that the parsing tree always begins by analyzing the input string rather than the set of productions.[1] In addition, the sub-routine is "predictive" in utilizing the productions to and establishing the next syntactic element.

Syntax of the Predicate Forms (A2)

The automaton A2 has as its domain the strings of L2. However, its syntax is based on Reichenbach's methods of linguistic analysis. In this section we will define a convenient formalism, the predicate form, and discuss its syntax. Later we will discuss how the processor discovers the L3 (predicate function) mapping of an L2 string. In defining the syntax C of A2, it will be shown that $\Gamma 1$ was designed in order to simplify

[1] For a review of current parsing algorithms see Bobrow.

the mapping into a predicate form. As in Γ1, the patterns that can be specified by a predicate form depend on the verb. Thus, the forms fall into four basic categories; equational, intransitive, transitive and factitive forms.

Equational Forms -	
PRED(ARG) Examples:	John is home. John is tall.
PRED (φ)	There is a man.
PRED(ARG,ARG)	John is taller than Peter.
Intransitive Forms -	
PRED(ARG)	The Dodgers win.
	The Dodgers win seldom.
Transitive Forms -	
PRED(ARG,ARG)	Tall John loves Mary.
PRED(ARG,ARG,ARG)	John saw Peter at the track.
Factitive Forms -	
PRED(ARG,ARG,ARG)	John elected Peter the chairman.

With one exception the verb types used in the above classification follow conventional definitions. However, following Sledd, factitive verbs are also included. Factitive verbs are transitive verbs that take an object complement.

The following predicate functions show the L3 mappings of the examples. In order to avoid using Church's Lambda notation to bind the variables, the convention of using upper case letters for the nonterminal elements and following them by the variables in lower case letters, is utilized to fully define the predicate function.

(i) PRED is home (ARG John)

(ii) PRED is tall (ARG John)

(iii) PRED is a man (ARG∅)

(iv) PRED is taller than (ARG John, ARG Peter)

(v) PRED win (ARG The Dodgers)

(vi) PRED win seldom (ARG The Dodgers)

(vii) PRED loves (ARG Tall John, ARG Mary)

(viii) PRED saw at the track (ARG John, ARG Peter)

(ix) PRED elected (ARG John, ARG Peter, ARG the chairman)

One special characteristic of the mapping should be noted. It is not necessary that elements be contiguous for them to be bound to the same variable. Thus, the verb "saw" and the preposition "at the track" are not contiguous in the string yet appear so in the function. This characteristic of the syntax has influenced the design of the processor, as will be made explicit in a later section.

Using the syntax C shown in Table 3, and the same conventions for

Syntax C for Predicate Forms

1) PRED → PRED (PREDMOD)

2) PREDMOD → PREDMOD, PREDMOD

3) ARG → ARG (ARGMOD)

4) ARGMOD → ARGMOD, ARGMOD

Table 3

binding the variables, results in the following predicate functions for the previous examples:

(i)-(iv) identical

(v) PRED win (ARG Dodgers (ARGMOD the)

(vi) PRED win (PREDMOD seldom) (ARG Dodgers (ARGMOD The)

(vii) PRED loves (ARG John (ARGMOD Tall), Mary)

(viii) PRED saw (PREDMOD at the track)(ARG John, ARG Peter)

(ix) PRED elected (ARG John, ARG Peter, ARG chairman (ARGMOD the))

The mapping from L2 to L3 has not been formalized by the syntax C. However, this syntax is implicit in the processor and will be described in the same section.

Description of the Canonical Form Processor (A2)

The predicate forms have been designed to mechanize efficiently the problems of pattern recognition and of equivalence of strings by providing a limited number of canonical forms or patterns to describe a large number of natural language strings. The syntax implicit in the processor for canonical reduction is quite simple as is shown in Table 4. It should be noted that the mapping presupposes a description in L2. Another implication is the necessity to order the arguments. The ordering of arguments is not made explicit by the rewriting rules given; however, the ordering is implicit in the processor. The rule followed in ordering arguments is simply defining each one as it is found in a left to right scan of the L2 description.

1) S $\rightarrow$ PRED(ϕ)
PRED(ARG)
PRED(ARG, ARG)
PRED(ARG, ARG, ARG)

2) NP $\rightarrow$ ARG

3) VMEQ $\rightarrow$ PRED

4) VMITR $\rightarrow$ PRED

5) VMTR $\rightarrow$ PRED

6) VMEQ + ADJC $\rightarrow$ PRED

7) VFAC $\rightarrow$ PRED

8) ADJ $\rightarrow$ ARGMOD

9) THAN (deleted)

10) PREDEQ $\rightarrow$ ARG

11) PADV $\rightarrow$ PREDMOD

12) PRP $\rightarrow$ PREDMOD

Table 4

The flow diagram of F0, designed to behave like the automaton A2, is described in Figure 4. Although the syntax does not give a complete description of how the L2 to L3 mapping should be carried out, it will become clear in the descriptions of the subroutines. F1 is essentially a hypothesis generator. It examines the L2 input and decides on an appropriate canonical form. Should it find the string L2 to have an equational verb, the possible canonical forms are:

PRED(ϕ)

PRED(ARG)

PRED(ARG, ARG).

Intransitive verbs restrict us to the form:

PRED(ARG)

When the string has a transitive verb, we choose between the canonical forms:

PRED(ARG,ARG)

PRED(ARG,ARG,ARG).

Finally problem strings with factitive verbs must follow the form:

PRED(ARG,ARG,ARG)

Sub-routine F1 searches the string and locates the main verb. The verb class is noted in order to establish the appropriate forms. When no verb is located, control is transferred to F10, which notifies the programmer of the difficulty and stops. Once a verb has been located F11 generates a predicate form. F12 copies the form as the current prediction. The next sub-routine is F2; it binds the words of the problem string to the form. Thus, the words of each NP are bound to an ARG in accordance with a left-to-right scan of the problem string. When a one-to-one correspondence is established between the NPs and the ARGs the processor transfers to F14. F14 leaves all the names of the ARGs on a pushdown list. The next sub-routine is F13 which tests whether the pushdown list string named by the ARG is empty. Should the list be empty F6 is the next sub-routine; otherwise it is F4. F4 tests whether there are any variables beside an N or PN in the ARG named on the pushdown list. If there are not the processor returns to F13. When additional words are found F5 rewrites the predicate form as

$\text{ARG} \rightarrow \text{ARG(ARGMOD)}$

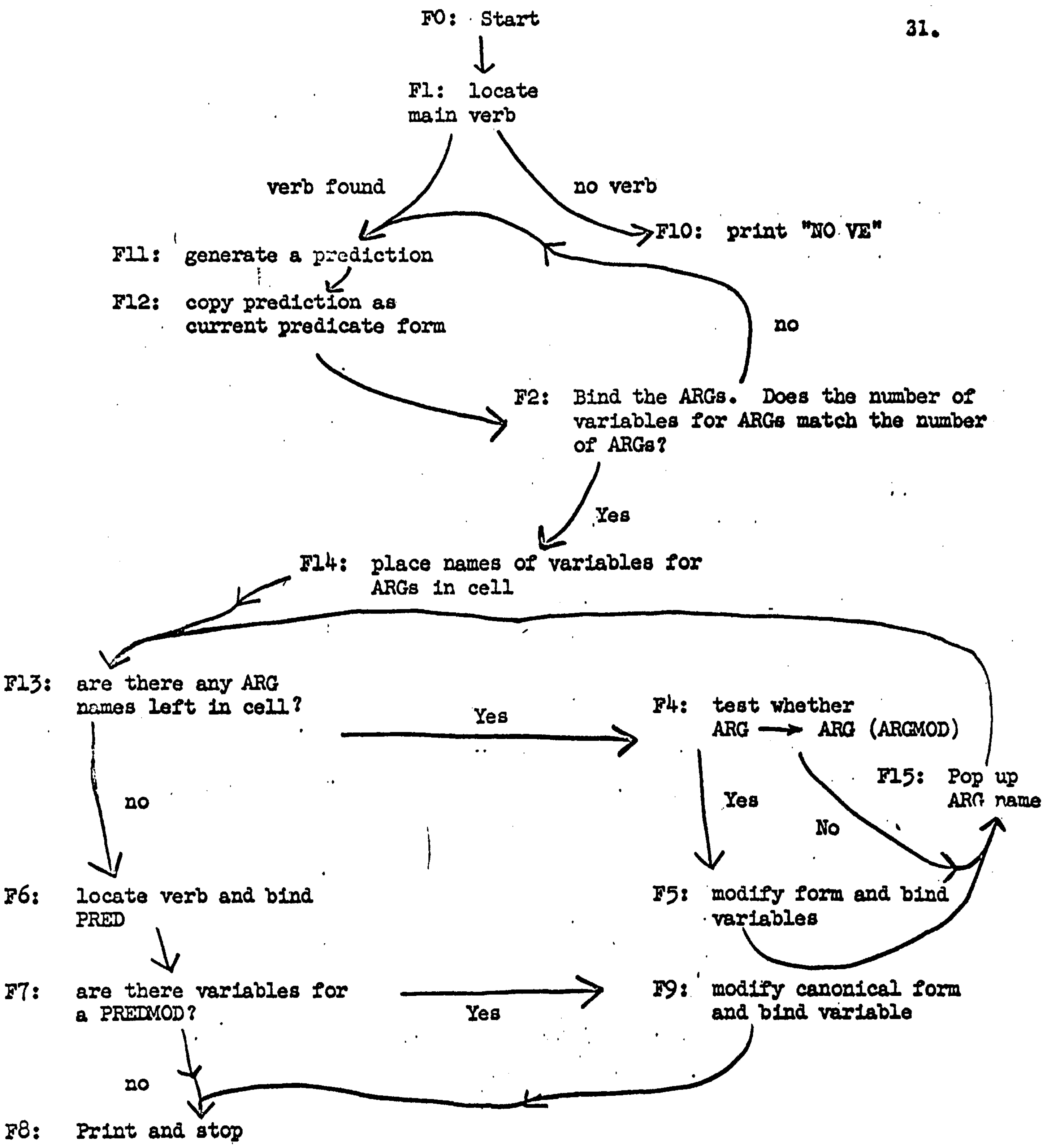

F0: Start
F1: locate main verb
verb found
no verb
F10: print "NO VE"
F11: generate a prediction
F12: copy prediction as current predicate form
no
F2: Bind the ARGs. Does the number of variables for ARGs match the number of ARGs?
Yes
F14: place names of variables for ARGs in cell
F13: are there any ARG names left in cell?
Yes
F4: test whether ARG ⟶ ARG (ARGMOD)
F15: Pop up ARG name
no
Yes
No
F6: locate verb and bind PRED
F5: modify form and bind variables
F7: are there variables for a PREDMOD?
Yes
F9: modify canonical form and bind variable
no
F8: Print and stop

FIG. 4

and erases the additional variables from the ARG and binds them to the ARGMOD. Following the execution of F5 the processor returns to F13. F6 locates the verb. For transitive, intransitive, and factitive verbs all the words in VTR, VITR and VFAC are bound to the PRED of the form. For equational verbs, the processor searches to see if it is followed by an ADJC or a PRP; if it is, the ADJC or a PRP becomes part of the PRED. F7 searches for a PADV or a PRP on the tree of a PREDTR. The words named by the PADV or PRP are bound to the PREDMOD. Sub-routine F8 then prints the L3 mapping of the problem string and halts the processor.

The following example illustrates the flow of the program:

Input $\sigma_1 \in$ L2 = (S(NP(ADJ Big)(N John)) (VP(VEQ(VMEQ is))(PREDEQ (PADJC(ADJC smarter)(THAN than)(NP(N Paul))))))

F1: locates the main verb "is". The available predicate forms are:

PRED(ϕ)

PRED(ARG)

PRED(ARG,ARG)

F11: The form PRED(ϕ) is generated.

F12: PRED(ϕ) is the current form.

F2: Since NP "Big John" is localized this predicate form is not appropriate. The executive returns to F11.

F11: The form PRED(ARG) is generated.

F12: PRED(ARG) is the current form.

F2: Since the NPs "Big John" and "Paul" are localized this form is inappropriate. Control returns to F11.

F11: The form PRED(ARG,ARG) is generated.

F12: PRED(ARG,ARG) is the current form.

F2: The NPs are in one-to-one correspondence with the ARGs. The variables are bound as

PRED (ARG Big John, ARG Paul)

and the executive transfers to F14.

F14: The names of the ARGs are placed in a pushdown list.

F13: Since the pushdown list is not empty control passes to F4.

F4: The first ARG in the pushdown list names "Paul". There is no ARGMOD so control passes to F15.

F15: Pops up the ARG naming "Paul".

F13: There is still an ARG name on the pushdown list.

F4: The ARG names "Big John", so the output becomes

PRED(ARG Big John (ARG Mod)), ARG Paul)

and then the variables are rearranged as

PRED(ARG John (ARGMOD Big)), ARG Paul).

F15: Pops up the last ARG name.

F13: Since the pushdown list is empty the executive program calls F6.

F6: Since L2 has a VEQ the PRED is bound as

PRED is (ARG John (ARGMOD Big), ARG Paul)

and then a further search is made for an ADJC or PRP. The ADJC naming "larger" is found so the predicate function becomes PRED is larger (ARG John (ARGMOD Big), ARG Paul).

F7: Since a PADV cannot be located and the verb is not transitive (so there can be no PREDTR) the processor calls sub-routine F8.

F8: The predicate function is printed and the processor halts.

<u>Recognition of Equivalent Strings (A3)</u>

Meaning equivalence is determined by A3 which attempts to apply a set of heuristically determined transformations in order to eliminate the differences between the strings σ_i and σ_j. The set of transformations TO was chosen on the basis that it is found useful in a large class of problems taken from Stoll. The set TO does not correctly solve all premise equivalence problems. Some examples will be given where it is inadequate.

The recognition of meaning equivalence is postponed until the mapping to L3 is complete. L3 was chosen to determine the pattern classes because the language not only orders the structure of L2, but also shows the dependencies between the elements of the language, and permits us to manipulate easily the L3 representations of σ_i and σ_j.

The actual recognition of equivalence is determined by the set of transformations TO.

Definition: The strings σ_1 and $\sigma_2, \in L_3$ are said to be "meaning equivalent" when we can find:

$$(T_i(T_j\ldots(T_m(\sigma_1))) = \sigma_2$$

where the T_i, T_j, ...T_m belong to the set TO. Where:

$$TO = \{ TPRN,\ TIMP,\ TTIME,\ TSYN \}$$

The domain and the range of TPRN are the ARGs of the predicate forms. The transformation replaces the current ARG with the corresponding one of the preceding premise. A necessary condition for the application of TPRN is that the first ARG be a pronoun in its L2 representation. For example, let:

σ_1: John loves music.

σ_2: He dressed quickly.

Their representation is

PRED loves (ARG John, ARG music)

PRED dressed (PREMOD quickly)(ARG He)

The transformation TPRN(σ_2) results in

PRED dressed (PREDMOD quickly)(ARG John)

The implied transformation, TIMP, has a domain of the predicate functions with a null argument. The transformation replaces the missing argument with that of the preceding premise. For:

σ_1: The Dodgers won the pennant.

σ_2: lost the series

with a representation of

PRED won (ARG Dodgers (ARGMOD the), ARG Pennant(ARGMOD the))

PRED lost (ARG ϕ, ARG series (ARGMOD the)).

TIMP (σ_2) results in the predicate function

PRED lost (ARG Dodgers, ARG series (ARGMOD the)).

The time transformation, TTIME, has as its domain the predicates. The range is also the predicates. This transformation eliminates auxiliary verbs and replaces the main verb with its root. The main verb is determined by the L2 representation of the string. An example would be:

σ_1: John should go home.

with an L3 representation

PRED should go (ARG John, ARG home)

Thus TTIME (σ_1) becomes

PRED go (ARG John, ARG home)

The synonym transformation, TSYN, has a domain of the words $W_i \in$ L2. Its range is also the words $W_i \in$ L2. The transformation is defined by replacing any W_i by its synonym as defined in the dictionary of the processor. The effect of TSYN can be seen on $\sigma_1 \in$ L1.

σ_1: John is happy.

which has an L3 representation

PRED is happy (ARG John)

after TSYN(σ_1) the predicate function might appear as

PRED is glad (ARG John)

This approach can certainly lead to difficulties.

Some problems in semantics have been avoided. A word can take on various meanings depending on the context, as in:

The <u>bug</u> crawled along the leaf.

The <u>bug</u> in the program was found.

He likes to <u>bug</u> me.

The word <u>bug</u> takes on a different meaning in each sentence. The mistakes that transformations can lead to should be evident. In some contexts the TSYN might be appropriate while in others it is not.

Another type of difficulty that has not been considered in the derivation of meaning equivalent strings is the following: One possible transformation contracts a number of arguments in the L3 representation of a string. Thus, σ_1, $\sigma_2 \in$ L2.

σ_1: John hits the ball with the bat.

σ_2: John bats the ball.

would have their respective representations as follows in L3:

PRED hits (ARG John) ARG ball (ARGMOD the), ARG bat (ARGMOD the))

PRED bats (ARG John, ARG ball (ARGMOD the))[1]

By changing the predicate, a 3 ARG function becomes a 2 ARG function with the same meaning. By working with the set TO, the great majority of problems in Stoll are amenable to solution. However, the processor is not capable of doing justice to the human abilities of linguistic resolution. One noticeable characteristic of utilizing TO as a recognition device is its tendency to err by not recognizing equivalent strings rather than by unjustified recognition.

Although this section defines the scope and effect of TO, it is also necessary to specify under what conditions the automaton attempts to apply one of the transformations, and under what conditions the processor will stop trying to match the strings. The criteria for applying a member of TO, and the decision to halt, will be made explicit in the next section.

[1] Example thanks to D. Kuck.

Structure of the Equivalence Recognizer (A3)

The flow chart (see Figure 5) of G0 was intended to implement A3. Clearly, meaning equivalence, as defined by G0, can only be understood in light of the problem context. Thus, in the formalization of the sentential calculus, we shall consider $_1$ and $_2$

$_1$: John will go home.

$_2$: John went home.

as meaning equivalent, because in this problem context meaning is time invariant. Obviously this is not true in conversational English. The program G0 initially calls G1 whose function is to test the number of ARGs in the problem strings. Failing to find the number of ARGs to be the same, control is passed to G3. G3 is one of a set of sub-routines, including G13, G17, G14 and G15, designed to notify the programmer that the strings were not found to be meaning equivalent and briefly indicate the reason. Should the problem strings have the same number of ARGs control is passed to G4 which tests for equality of PREDs. When this requirement is not met G5 is executed by dropping any VAUX and attempting to find the root of the main verb. If the existing differences are not eliminated by G5 the executive transfers control to G6. This sub-routine, like G22, G21 and G20, attempts to eliminate the differences between strings by using a dictionary search. Sub-routine G7 tests the PREDMODs for equality. When any differences in the PREDMODs are reconciled the executive program calls G8. It also tests for identity in the sub-strings. In this case the matching is of the first ARG of each string, the second ARG of each string, etc...until a

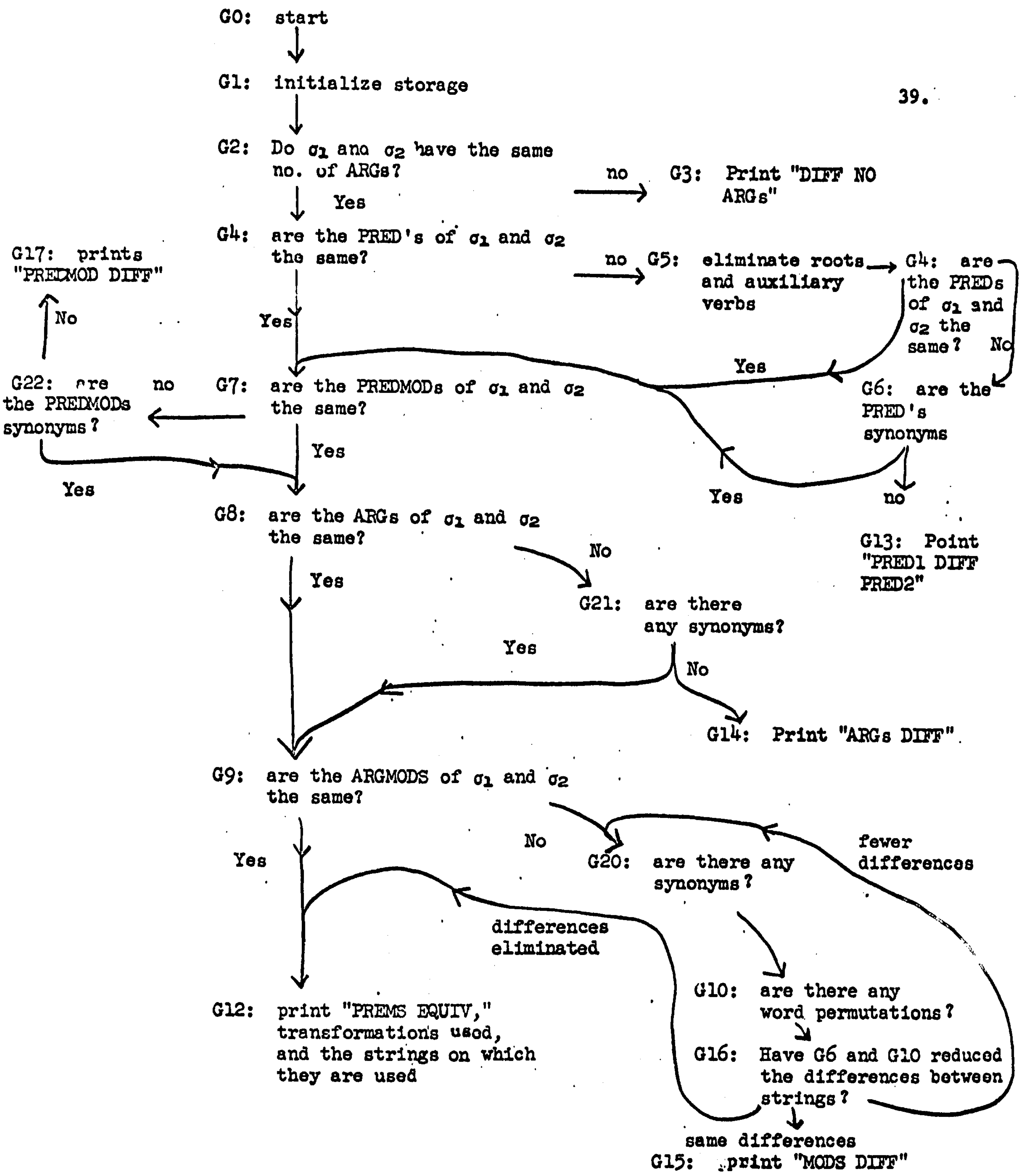

G0: start
G1: initialize storage
G2: Do σ_1 and σ_2 have the same no. of ARGs?
no
G3: Print "DIFF NO ARGs"
Yes
G4: are the PRED's of σ_1 and σ_2 the same?
no
G5: eliminate roots and auxiliary verbs
G4: are the PREDs of σ_1 and σ_2 the same?
No
Yes
G6: are the PRED's synonyms
Yes
no
G13: Point "PRED1 DIFF PRED2"
G17: prints "PREDMOD DIFF"
No
G22: are the PREDMODs synonyms?
no
G7: are the PREDMODs of σ_1 and σ_2 the same?
Yes
Yes
G8: are the ARGs of σ_1 and σ_2 the same?
No
Yes
G21: are there any synonyms?
Yes
No
G14: Print "ARGs DIFF".
G9: are the ARGMODS of σ_1 and σ_2 the same?
No
Yes
G20: are there any synonyms?
fewer differences
differences eliminated
G10: are there any word permutations?
G16: Have G6 and G10 reduced the differences between strings?
same differences
G15: print "MODS DIFF"
G12: print "PREMS EQUIV," transformations used, and the strings on which they are used

FIG. 5

difference is found in the strings. A difference in the strings leads the processor to execute G20, G10, and G16. As previously mentioned, G20 searches for synonyms. G10 attempts to reduce differences by finding permutations of the differing ARGMODs. Finally, G16 keeps track of the number of differences in the strings (based on the order and symbols on each ARGMOD list). When all differences are eliminated control is passed to a print routine, G12. Should the number of differences remain constant on successive executions of the G20, G10 and G16 loop, the processor calls sub-routine G15. If the number of differences is decreasing the loop is repeated.

The following example illustrates the logic of the system:

σ_1: PRED is (ARG John(ARGMOD Big tall), ARG home)

σ_2: PRED is (ARG John (ARGMOD Tall large), ARG home)

G0: Calls G1.

G1: Initializes storage.

G2: Both σ_1 and σ_2 have two ARGs so the executive calls G4.

G4: Since both the ARGs have the PRED "is" control is transferred to G7.

G7: There are no PREDMODs so the processor continues to G8.

G8: ARGs are checked in order, first σ_1 and σ_2 are shown to have the same ARG "John", then the second ARGs are both identified as "home". Since no difference exists the processor calls G9.

G9: In the first ARGMOD the difference count is 2 since "Big tall" and "Tall large" are both different symbols. No second ARGMOD is located for either σ_1 or σ_2. The executive program calls G20.

G20: Attempts to locate "Tall" as a synonym for "big" and "large" as a synonym for "tall", and fails in both cases.

G10: Notes that the difference count can be decreased by rearranging the ARGMODs as "Big tall" and "Large tall".

G16: Since the number of differences has decreased from 2 to 1 the executive returns to G20.

G20: This time the synonym "Large" is located for "Big" (assuming that the synonym is stored in the dictionary DO).

G10: Since no differences are located by G10 it cannot perform any permutations.

G16: The differences between the ARGMODs of σ_1 and σ_2 have been eliminated so a transfer is made to G12.

G12: The print out "PREMS EQUIV" is followed by the fact that the transformation TSYN was necessary on "Big" and TPERM on "Tall large".

Summary

This completes our description of a processing system for problems in the statement calculus. The system accepts problems as they are normally written in English and attempts to produce a formalized equivalent as its output. It makes uses of a series of automata, the first of which attempts to identify the elementary premises and the logical connectives. Two additional automata are used in order to compare premises and to determine whether or not they should be identified as equivalent. As a first step, each premise is mapped into a canonical form which simplifies the identification of equivalent premises. In the second step, pairs of premises are compared. This automata makes use of a number of meaning-preserving transformations. In a sense, two premises are equivalent if one can be derived from the other with the aid of these transformations. Otherwise, the premises are evaluated as not equivalent. Although this processor is limited to a particular class of problems, it was designed with two purposes in mind: as an attempt to simplify the problems of communications between programmer and computer and to clarify those processes by means of which meaning is extracted from natural language.

GLOSSARY OF ABBREVIATIONS

h.i.s.	hypothesized input string
i.s.	input string
l.h.s.	left-hand side
n.w.f.s.	not a well formed string
p.o.s.	part of speech
r.h.s.	right-hand side

REFERENCES

Bobrow, D., "Syntactic Analysis of English by Computer - A Survey", Proceedings of the FJCC, 1963.

Carnap, R., Introduction to Symbolic Logic and its Applications. New York: Dover, 1958.

Chomsky, N., Syntactic Structures. 's-Gravenhage: Mouton, 1957.

Green, B. F., Jr., Wolf, A. K., Chomsky, C., and Laughery, K., "Baseball: an Automatic Question-Answerer". Proceedings, Western Joint Computer Conference. May, 1961, pp. 219-224.

Krulee, G. K., Kuck, D. J., Landi, D. M., and Manelski, D. M., "Natural Language Inputs for a Problem-Solving System". Behavioral Science. July, 1964, pp. 281-288.

Kuck, D. J., and Krulee, G. K., "A Problem Solver with Formal Descriptive Inputs". Computers and Information Science. Baltimore: Spartan, 1964, pp. 344-374.

Manelski, D. M., "A Heuristic Approach to Natural Language Processing", Unpublished Ph.D. thesis, Northwestern University, 1964.

Newell, A. (Ed.), Information Processing Language-V Manual. Englewood Cliffs: Prentice-Hall, 1961.

Newell, A., and Shaw, J. C., "Programming the Logic Theory Machine". Proceedings, Western Joint Computer Conference. February, 1957, pp. 230-240.

Reichenbach, H., Elements of Symbolic Logic. New York: MacMillan, 1938.

Sledd, J., A Short Introduction to English Grammar, Chicago: Scott, Foresman, 1959.

Stoll, R. R., Sets Logic and Axiomatic Theories. San Francisco: W. H. Freeman, 1961.

Walker, D. E., and Bartlett, J. M., "The Structure of Language for Man and Computer: Problems in Formalization", Information System Science and Engineering. New York: McGraw-Hill, 1963.