



This blank page was inserted to preserve pagination.

A CANONIC TRANSLATOR

by

Joseph Wright Alsop

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1967

Signature of Author

Department of Electrical Engineering
19 May 1967

Certified by

Thesis Supervisor

Accepted by

Head of Department

Acknowledgement

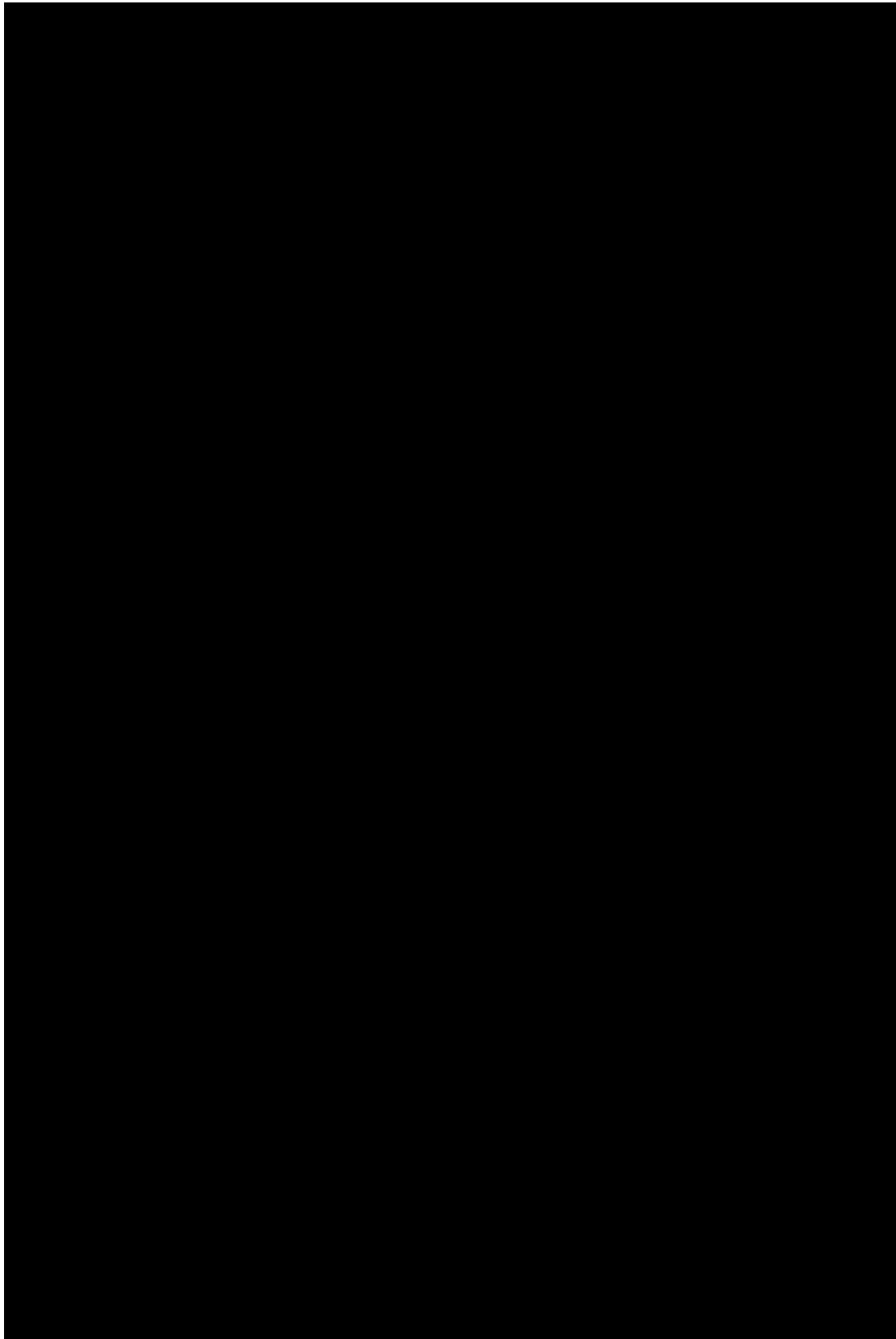
A heartfelt note of thanks is due my thesis supervisor. Not only has Professor Donovan's work provided the motivation for the work herein described; his suggestions, ideas and enthusiasm have made possible a successful conclusion to this thesis.

I am indebted to Professor Joseph Weizenbaum for the use of the SLIP system and most particularly for the time he took to explain the details of its use.

Summary

An algorithm to recognize and translate sets of character strings specified by canonic systems is presented. The ability of canonic systems to define the context sensitive features of strings and to specify their translation allows the algorithm to recognize and translate real computer languages. It is also applicable in other language systems.

Canonic systems are discussed, and several examples of their use are given. The algorithm is described, and examples of canonic translation are presented using a program which implements it.



Figures and Diagrams

Structure of program.	18
Illustration of scan.	22
Flowchart of simplified algorithm.	23
Flowchart of general algorithm.	28

A Canonic Translator

The development of a generalized compiler whose function is directed by a formal language specification has aroused significant interest and effort. This thesis presents an algorithm for the recognition and translation of character strings belonging to a set of strings whose syntax and translation have been defined by a canonic system. Since these systems are capable of defining context sensitive features of language, the algorithm can recognize and translate real computer languages. It is applicable to an even wider class of language systems, including boolean algebras and theorem proving, which can be characterized by this method.

Canonic systems form the basis and motivation for this work. The first task of the paper is to discuss briefly and informally the improved specification of syntax and translation made possible by the development of canonic systems. The discussion includes a description of the form of the systems and several examples, among them a complete formal description for the syntax of the string processing language SNOBOL. The contribution of this thesis lies in the presentation of an explicit algorithm which employs a canonic system characterizing the syntax and translation of a set of source strings to recognize a particular source string and perform the translation. The latter part of the thesis describes the algorithm and the program which implements it.

I. Formal Syntax Specifications

Backus-Naur Form is the most widely known formal specification of syntax. It provides a convenient starting point for a discussion of canonic systems. The general form of a rule or production of a BNF specification is as follows:

$$\langle \text{name 1} \rangle ::= \text{terminal 10} \langle \text{name 11} \rangle \dots \langle \text{name 1n} \rangle \text{terminal 1n} \mid \text{terminal 20} \langle \text{name 22} \rangle \dots \langle \text{name 2m} \rangle \text{terminal 2m} \mid \dots$$

The sign ::= should be read "may be replaced by" and the vertical bar represents "or". The names enclosed within brackets are arbitrary designations for defined sets of strings. The definition may be recursive; that is, the set on the left may be defined in terms of itself if the name of the set also appears on the right. "terminal n m" designates an arbitrary string of terminal characters, possibly the null string. As a concrete example, consider the following BNF system.

$$\begin{aligned} \langle \text{assignment} \rangle & ::= \langle \text{letter} \rangle = \langle \text{expression} \rangle \\ \langle \text{expression} \rangle & ::= \langle \text{letter} \rangle \mid \langle \text{letter} \rangle + \langle \text{expression} \rangle \\ \langle \text{letter} \rangle & ::= X \mid Y \mid Z \end{aligned}$$

An example of a string which is a member of the set <assignment> is:

$$Y = X + Z$$

The strings comprising a set defined by a BNF system normally appear to be generated in a "top-down" manner. The highest level definition (<assignment>) is generally placed first, and one normally reads a BNF rule from left to right. In order to gain some insight into the form and nature of canonic systems without launching into a formal definition, consider turning a BNF production around and modifying the punctuation somewhat.

$$1. \quad v \text{ letter} \vee x \text{ expression} \vdash v = x \text{ assignment}$$

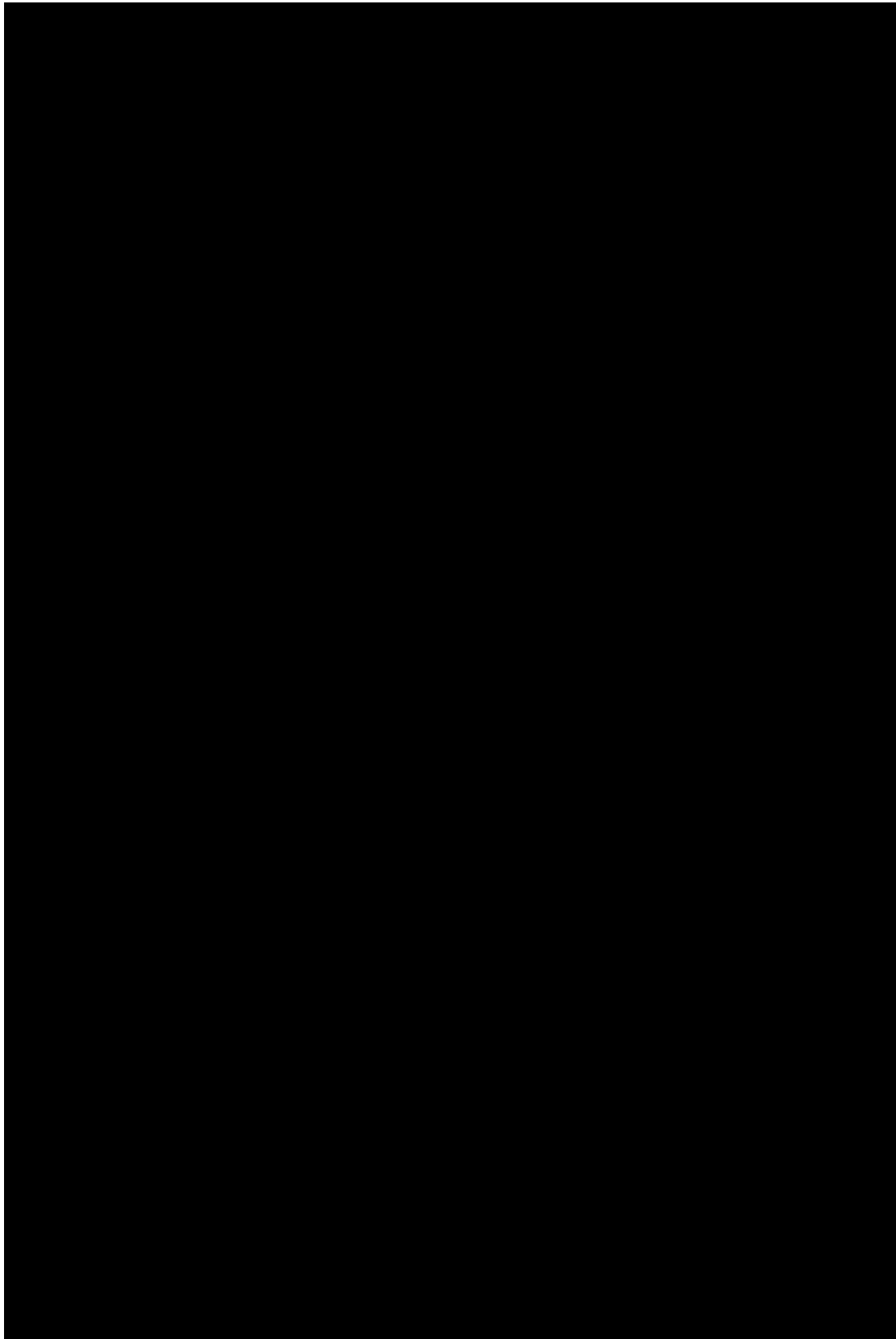
The lower case letters (v and x) are variables representing strings

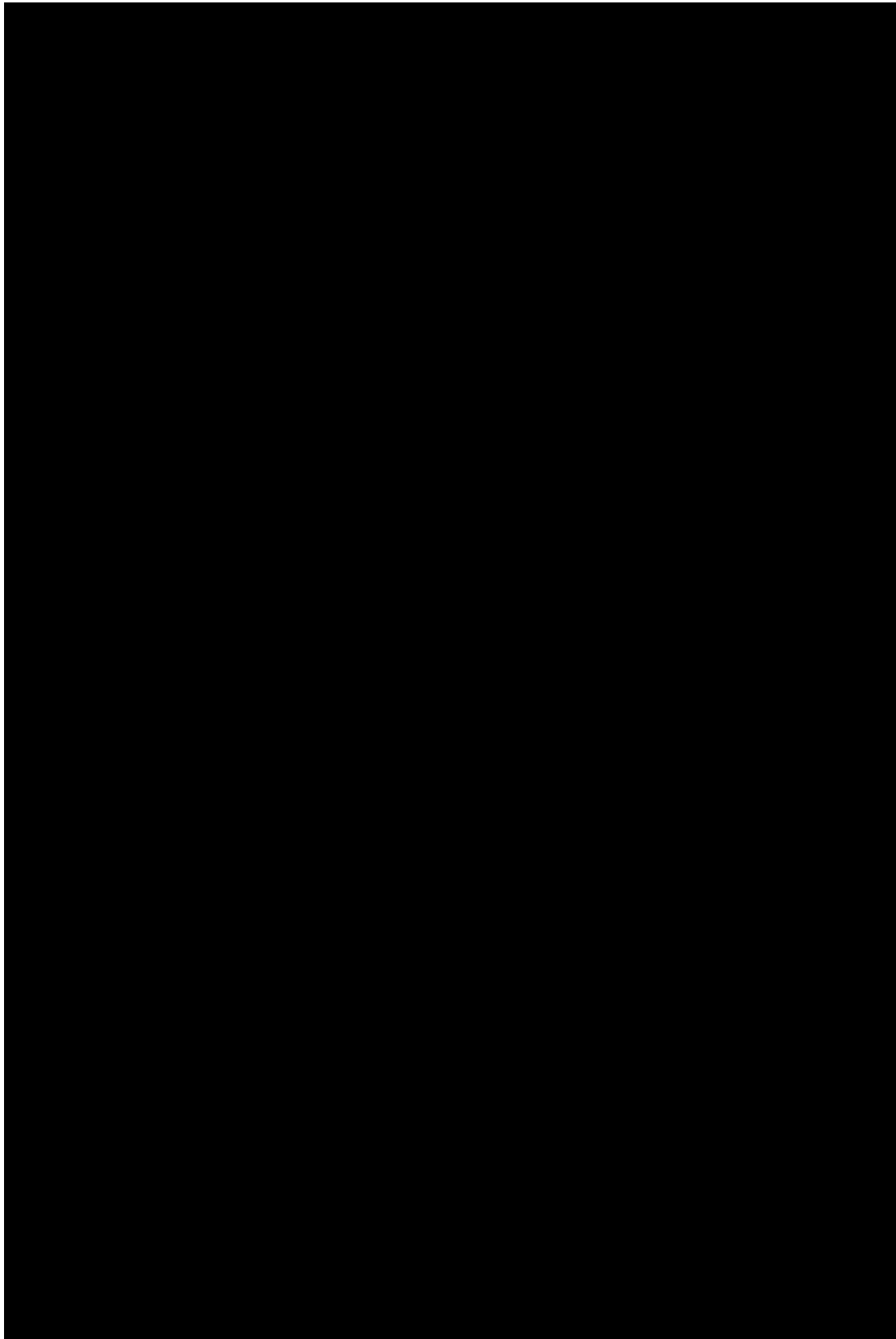
chosen from their respective sets (letter and expression). The names of the sets are underlined and called predicates. The definition may be read very elaborately as follows: "If v represents a string chosen from the set letter, and if x represents a string chosen from the set expression, then the string formed by concatenating the string represented by v with an equals sign and the string represented by x is a member of the set assignment." The sign \wedge acts as the conjunctive "and", and the sign \vdash acts as an assertion sign. A string of variables and terminal characters (e.g. $v=x$) is a term, and a term followed by a predicate in the manner above is a remark. Those remarks to the left of the assertion sign are referred to as premises; those to the right as conclusions. This example illustrates the most basic form of a canon in a canonic system. A more formal description may be found in Donovan (2) and Donovan and Ledgard (3). This discussion will remain highly informal.

What improvements in the definition of a syntax do canonic systems permit? The principal weakness of BNF systems is their inability to describe the context sensitive features of a set of strings; for example, the requirement in most computer languages that all reference labels of a program be singly defined as statement labels. This restriction could only be imposed in BNF notation by some process akin to defining each possible legal program, in toto, in a separate BNF rule. Certainly all sets of strings which can be defined in BNF may be defined by a canonic system by transforming the rules in the manner illustrated above. In addition, one may "cross-reference", or use a variable more than once on the left.

$$2. \quad x \text{ name } \wedge x \text{ label } \vdash x \text{ labelname}$$

Labelname is the intersection of label and name; that is, only those strings which are members of both the set label and the set name are members





and makes it possible to generate all ordered pairs with the property described.

A concrete example of the production of a particular member of a defined set will perhaps serve to clarify the nature and recursive properties of canonic systems. Assume we wish to show that $\langle A \langle X, Y \rangle \rangle$ is a member of the set notin. Using canon 4, we may assert

A letter.

We may then substitute this result into the premise of canon 7, and assert that

$\langle A \langle \wedge \rangle \rangle$ notin.

We then derive from canon 5 that

$\langle A \langle X \rangle \rangle$ differ

$\langle A \langle Y \rangle \rangle$ differ.

Finally, we apply canon 8 twice as follows:

$\langle A \langle \wedge \rangle \rangle$ notin $\not\Leftarrow$ $\langle A \langle X \rangle \rangle$ differ \vdash $\langle A \langle X, \rangle \rangle$ notin
 $\langle A \langle X, \rangle \rangle$ notin $\not\Leftarrow$ $\langle A \langle Y \rangle \rangle$ differ \vdash $\langle A \langle X, Y, \rangle \rangle$ notin

Note that we use the conclusion from the first application of the canon to establish the premise in the second application.

Now that the reader has grasped some of the power and elegance of canonic systems, a short history of their development is in order. This work is based completely upon the presentation of canonic systems by Donovan and Ledgard (3) and Donovan (2), who is responsible for their appearance in present form. His work evolved from an applied variant of Smullyan's elementary formal systems (6) and Post's canonical systems (4). The present canonic systems are so named in recognition of Post's work.

To further illustrate canonic systems, I present a complete syntactic definition of a restricted computer language MINI MAD. The present example and the foregoing example of notin both draw heavily from the examples

presented in Donovan (2).

MINI MAD will permit only a few principal types of statements: an assignment statement, a transfer statement, and a statement formed by combining a simple conditional with one of the two other statements. All programs must terminate with an unlabeled END OF PROGRAM statement. The only boolean operator allowed is arithmetic equality (.E.), the only arithmetic operator allowed is addition (+), and only arbitrary length integers will be permitted as constants. The permissible statement labels are the single letters A, B and C; the variable names allowed are the letters X, Y and Z. In addition, restrictions on statement length will be omitted and no blanks will be allowed save those which are part of the statement definition (e.g. TRANSFER TO). The character * will be adopted as an end-of-card character, analogous to a carriage return. It should be understood that all restrictions and omissions are introduced for the sake of simplicity. A complete formal syntactic definition of the string-processing language SNOBOL may be found in appendix 2.

The following example is a member of the set MINI MAD program, with a carriage return substituted for the character *.

```
A  X=15
B  X=X+1
   WHENEVER X .E. 123, TRANSFER TO A
   TRANSFER TO B
```

Three canons will suffice to define the set of arbitrary length integers.

9. $\vdash 0 \Delta 1 \Delta 2 \Delta \dots \Delta 8 \Delta 9 \text{ digit}$
10. $d \text{ digit} \vdash d \text{ integer}$
11. $d \text{ digit} \not\vdash i \text{ integer} \vdash di \text{ integer}$

The use of the predicate notin, defined previously, will later implement the restriction that no statement labels be multiply defined.

12. $\vdash \langle A \triangleleft B \rangle \triangleleft \langle A \triangleleft C \rangle \triangleleft \langle B \triangleleft C \rangle$ differ
13. $\langle x \triangleleft y \rangle$ differ $\vdash \langle y \triangleleft x \rangle$ differ
14. d digit $\vdash \langle d \triangleleft \Lambda \rangle$ notin
15. $\langle x \triangleleft y \rangle$ notin $\not\vdash \langle x \triangleleft d \rangle$ differ $\vdash \langle x \triangleleft d, y \rangle$ notin

One should keep in mind that only lower case letters are used as variables representing strings. The signs $\vdash, \not\vdash, \triangleleft, \triangleleft$ are punctuation signs in the canonic system itself. All other characters are drawn from the alphabet of the language being defined.

The definition of the predicate in will serve to implement the restriction that all reference labels be defined. The set in will consist of pairs of letter lists such that all letters in the first list appear somewhere in the second list. If the list of reference labels and the list of statement labels in a program satisfy this relationship, we know that there is at least one statement label corresponding to every reference label.

16. $\vdash A \triangleleft B \triangleleft C$ label
17. $\vdash \langle \Lambda \triangleleft \Lambda \rangle$ in
18. $\langle x \triangleleft y \rangle$ in $\not\vdash l$ label $\vdash \langle x \triangleleft l_1 y \rangle$ in
19. $\langle x \triangleleft y \rangle$ in $\not\vdash l$ label $\vdash \langle l_1 x \triangleleft l_1 y \rangle$ in
20. $\langle x \triangleleft y \rangle$ in $\not\vdash \langle z \triangleleft y \rangle$ in $\vdash \langle xz \triangleleft y \rangle$ in

Canon 17 provides a simple starting point for the recursive production of the more elaborate members of in, and corresponds to a program with neither statement nor reference labels. The next two canons describe the ways in which one may add to the lists of statement and reference labels. We may of course add a label at will to the list of statement labels, and may add a label to the reference label list as long as we also add it to the list of statement labels. The last canon provides for multiple referencing of a statement label. Using canons 16 through 19 alone, it is not possible to produce the following member of in

$\langle B, B, \langle A, B, C \rangle \rangle$

We may define the set expression as follows.

21. $\vdash XAYAZ$ variable
22. v variable $\vdash v$ expression
23. i integer $\vdash i$ expression
24. v variable ϕ x expression $\vdash v + x$ expression
25. i integer ϕ x expression $\vdash i + x$ expression

The predicate next defined, conditional, will permit us to transform any unconditional statement into a conditional statement when a string from the set is placed before the unconditional statement.

26. $\vdash \wedge$ conditional
27. x expression ϕ y expression \vdash WHENEVER $X.E. Y,$ conditional

Canon 26 allows us to produce a string which leaves the statement unchanged.

Canon 27 defines a set of strings which will change any unconditional MINI MAD statement (e.g. $X = 3$) into a conditional statement (e.g. WHENEVER $X + Y .E. Z, X = 3$).

The "building block" sets defined so far will permit us to define the set of MINI MAD programs in fairly short order. A convenient vehicle for the task is a predicate of order three. The first element of the ordered triplets which make up the set program with label lists will be a list, punctuated by commas, of all statement labels used. The third element will be a similar list of reference labels. The second element will be the string of statements in which these labels are used. Again, we begin with a convenient starting point for later recursion.

28. $\vdash \langle \wedge \langle \wedge \langle \wedge \rangle \rangle \rangle$ program with label lists
29. $\langle s \langle p \langle r \rangle \rangle \rangle$ program with label lists ϕ v variable ϕ x expression ϕ c conditional
 $\vdash \langle s \langle CV = X * p \langle r \rangle \rangle \rangle$ program with label lists
30. $\langle s \langle p \langle r \rangle \rangle \rangle$ program with label lists ϕ $\langle l \langle label \rangle \rangle$ ϕ v variable ϕ x expression ϕ
 ϕ c conditional ϕ $\langle l \langle s \rangle \rangle$ notin
 $\vdash \langle l \langle s \langle l \rangle \rangle \rangle$ $CV = V * p \langle r \rangle$ program with label lists

Canons 29 and 30 describe the way in which we may add an assignment statement, either conditional or unconditional. Using the first canon of the two, we may add an unlabeled assignment statement; using the second, we may add a labeled statement. Note that the use of notin in canon 30 imposes the restriction that the label used must not be in the list of previous statement labels.

30. $\langle s \langle p \langle r \rangle \text{ program with label lists } \phi \text{ label } \phi \text{ c conditional} \rangle$
 $\vdash \langle s \langle \text{ C TRANSFER TO } l^* p \langle l, r \rangle \text{ program with label lists} \rangle$
31. $\langle s \langle p \langle r \rangle \text{ program with label lists } \phi \text{ } l_{\Delta m} \text{ label } \phi \text{ c conditional } \phi \rangle$
 $\vdash \langle m \langle s \text{ notin } m_1 s \langle m \text{ C TRANSFER TO } l^* p \langle l_1 r \rangle \text{ program with label lists} \rangle \rangle$

These two canons allow use to construct strings which include labeled and unlabeled, conditional and unconditional transfer statements in a manner analogous to that of the preceding pair of canons. We now need but one more canon to produce strings which are legal MINI MAD programs.

32. $\langle s \langle p \langle r \rangle \text{ program with label lists } \phi \langle r \langle s \rangle \text{ in } p \text{ END OF PROGRAM* } \text{ MINI MAD program} \rangle \rangle$

This canon insures that all reference labels in the members of the set MINI MAD program are defined, and that all programs are properly terminated. This completes one of many possible canonic system definition or programs in MINI MAD. The canons are collected in sequence in appendix 1.

If the reader has clearly understood the manner in which canons may define, by production, the syntax of real computer languages, one further illustration may provide some insight into the manner in which these systems may also define translation. Assume one wishes to translate MINI MAD into another language, for instance an assembly language such as FAP. In order to accomplish this, one might expand program with label lists to include a fourth term which would contain the translation of the string of statements.

The canon for an unconditional, unlabeled TRANSFER TO statement might appear as follows.

$$33. \quad \langle s \langle p \langle r \langle t \rangle \text{ program with label lists and translation } \rangle \rangle \langle \mathcal{L} \text{ label} \rangle \\ \vdash \langle s \langle \text{TRANSFER TO } l^* \text{ } \langle \mathcal{L}_1 \text{ } \langle t \rangle \text{ } \text{TRA } l^* \rangle \rangle \text{ program with label lists and translation } \rangle$$

This possibility of canonic specification of translation will be pursued further in the description of the algorithm which forms the contribution of this thesis, to which I now turn.

II. The Recognition and Translation Algorithm

Canonic systems will prove very useful in explicitly and concisely defining sets of strings such as computer languages. Such definitions would eliminate many ambiguities existing in language manuals. These systems could prove of greater value, however, if a canonic system could be used as a basis for recognizing strings from the defined set. In addition, if the members of the defined set are ordered pairs, triplets, etc., the usefulness of canonic systems would be still further extended if the algorithm could be used to produce the missing terms corresponding to a given term. The remaining part of this thesis discusses such an algorithm, the program which implements it, and the nature of the constraints imposed on the canons in order that the program be able to interpret them.

This algorithm is an extension of the algorithm presented by Cheatham and Sattley (1), which is capable of recognizing strings produced by a Backus-Naur system. The modifications to their algorithm, which appears here in quite different form, reflect the greater power of canonic systems in defining strings. These modifications include mechanisms for handling predicates of degree greater than one, for properly interpreting the multiple use of a variable among the premises, and for generating the translation specified. In the case of a canonic system where all predicates are of degree one, and no "cross-referencing" is used, the algorithm operates in a manner almost identical to that of Cheatham and Sattley.

The program which embodies the algorithm divides into two parts. A preliminary phase checks the syntax of the canonic system used. It insures, for example, that all variables used in the conclusion of a canon are to be found in the premises, and that all predicates used as premises are defined somewhere as conclusions. Further restrictions, which will be clarified

later, are imposed on the form of the canons and are checked at this point. The program then assembles the canons into a list structure which reflects their form and content, and control is passed to the evaluative phase of the program. The SLIP list-processing system, developed by Weizenbaum (7) vastly simplified the implementation of the algorithm.

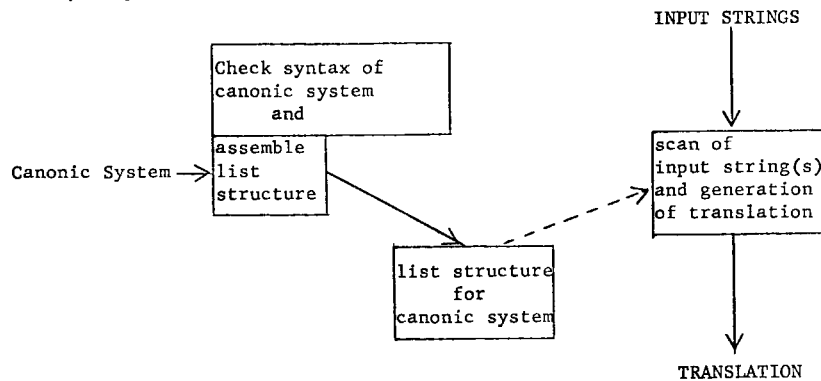


Fig. 1. Structure of Program

The second part of the program represents the principal programming effort. This phase scans the input string, determines whether it satisfies the canonic definition, and generates any associated translations. The algorithm is principally "top-down"; it attempts first to match the input string against the final predicate in the canonic system (e.g. MINI MAD program), and it arrives only through recursion at a lower-level predicate, (e.g. integer or digit). Consider the following simplified statement of the algorithm for the case of a canonic system involving only predicates of degree one. The simplified algorithm will be later expanded to include more general cases. Imagine an arbitrary character string, with a mental

pointer to the left of the first character, and a canonic system defining a set of strings. We wish to determine whether the character string is a member of the set.

1. The program considers in sequence those canons directly defining the string in question, and performs the following steps (2 through 6) for each such canon.
2. The conclusion of the canon is matched, item by item, against the input string. If the item in the conclusion is a terminal character, step 3 is performed; if a variable, step 4 is performed. If the end of the canon is reached, the algorithm proceeds to step 5.
3. The item in the conclusion is a terminal character. It is compared with the character in the input string to the right of the mental pointer. If they are identical, the program returns to step 2 to consider the next item in the conclusion, with the pointer shifted one position to the right. If not, the scan fails and the program returns to step 1 to consider any remaining canons for the string.
4. The item in the conclusion is a variable, and the program must operate recursively to determine the definition of the variable in terms of the input string. In other words, it must determine the number of characters from the input string, commencing with the character to the right of the pointer, which should be allotted to the definition of this variable. To accomplish this, the program assembles a new input string which is a copy of all input characters to the right of the pointer, and picks a predicate among the premises of the canon which contains the variable. After saving its present state, the program returns to step 1 to determine the definition of the variable by examining the canons defining the premise predicate chosen. If there is no response upon return, the scan fails and the program returns to step 1 to consider alternative definitions of the string.

If there is a response, the program compares it with the original input string to determine the definition of the variable and moves the mental pointer to its new position following the definition of the variable.

The algorithm returns to step 2.

5. The scan of the conclusion is complete, and the definitions, in terms of the input characters, of the variables appearing in the conclusion have been recorded. The algorithm now inspects the premises. Those premises used in step 4 to determine the definitions of the variables in the conclusion may already be asserted, since they were used to generate the definitions. However, a variable may appear twice in the premises, and we must insure that the string which forms the definition of the variable is a member of both sets. The algorithm forms an input string from the definition of the variable and operates recursively to determine if the other premise containing the variable is also true; i.e., if the string which is the definition of the variable is also a member of the second set named as a premise predicate. Upon return, if there is no response, the algorithm returns to step 1 to pursue alternatives as before. If there is a response, the program insures that the string has been fully scanned. If there are still more unchecked premises, it treats them in the same manner. After all such premises have been successfully verified, the simplified algorithm proceeds to the last step.

6. The results of the scan at this level, which constitute the response for the next higher level, are assembled. There are no results if the scan failed. Otherwise, they consist of the input string with the mental pointer resting at the point where the scan of the conclusion was completed. The algorithm now returns to step 1, if there are more canons directly defining the set of which the input string is possibly a member. Since

each canon could conceivably add to the results, the program must actually be equipped to handle multiple results and hence multiple responses at the next higher level, and check out each possibility. The example which follows will serve to clarify the problem. If there are no further canons, the program proceeds to step 7.

7. The program "pops" its state; that is, it returns to pick up where it left off at the next higher level. If the highest level has been reached, then the results are examined for a completely scanned input string. If such a response is found, the input string is a member of the originally defined set. If not, there exists a syntax error in the string. It is not clear that the set of all syntactically incorrect sets will be recognized by the algorithm. This recognition may be unsolvable in general. The algorithm is flowcharted below.

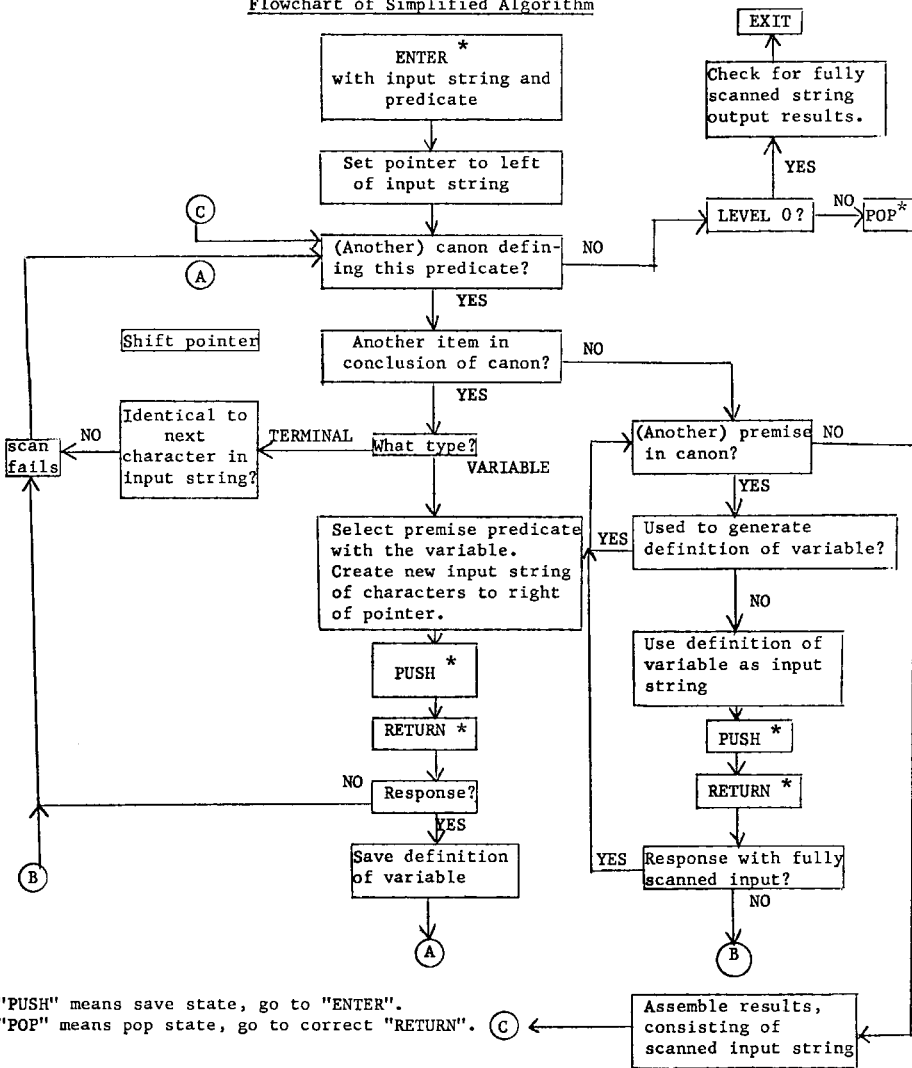
A simple example will serve to illustrate the process and the problems involved in multiple answers. Consider the following canonic system.

- 34. $\vdash 1 \text{ digit}$
- 35. $\vdash 2 \text{ digit}$
- 36. $\vdash 3 \text{ digit}$
- 37. $d \text{ digit} \vdash d \text{ integer}$
- 38. $d \text{ digit} \vee i \text{ integer} \vdash di \text{ integer}$

This system defines integers as arbitrary length strings of 1, 2 and 3. We wish to determine by use of the algorithm whether the string 31 is an integer. The process is described in the shorthand fashion below.

<u>Step</u>	<u>Recursion Level</u>	<u>Input String</u>	<u>Canon Considered</u>	<u>Result(s)</u>	<u>Next Action</u>
1	0	↓ 31	37	—	Push for <u>digit</u>
2	1	↓ 31	34	Fails	Next Canon
3	1	↓ 31	35	Fails	Next Canon
4	1	↓ 31	36	3 ↓ 1 <u>digit</u>	Pop
5	0	3 ↓ 1	37	3 ↓ 1 <u>integer</u>	<u>Next Canon</u>
6	0	↓ 31	38	—	Push for <u>digit</u>
7	1	↓ 31	34	Fails	Next Canon
8	1	↓ 31	35	Fails	Next Canon
9	1	↓ 31	36	3 ↓ 1 <u>digit</u>	Pop
10	0	3 ↓ 1	38	—	Push for <u>integer</u>
11	1	↓ 1	37	—	Push for <u>digit</u>
12	2	↓ 1	34	↓ 1 <u>digit</u>	Next Canon
13	2	↓ 1	35	Fails	Next Canon
14	2	↓ 1	36	Fails	Pop
15	1	1 ↓	37	1 ↓ <u>integer</u>	Next Canon
16	1	↓ 1	38	—	Push for <u>digit</u>
17	2	↓ 1	34	1 ↓ <u>digit</u>	Next Canon
18	2	↓ 1	35	Fails	Next Canon
19	2	↓ 1	36	Fails	Pop
20	1	1 ↓	38	—	Push for <u>integer</u>
21	2	↓ 1	37	—	Push for <u>digit</u>
22	3	↓	34	Fails	Next Canon
23	3	↓	35	Fails	Next Canon
24	3	↓	36	Fails	Pop
25	2	↓	37	Fails	Next Canon
26	2	↓	38	—	Push for <u>digit</u>
27	3	↓	34	Fails	Next Canon
28	3	↓	35	Fails	Next Canon
29	3	↓	36	Fails	Pop
30	2	↓	38	Fails	Pop
31	1	↓	38	1 ↓ <u>integer</u>	Pop
32	0	↓	38	31 ↓ <u>integer</u> 3 ↓ 1 <u>integer</u>	Done

Flowchart of Simplified Algorithm



"PUSH" means save state, go to "ENTER".

"POP" means pop state, go to correct "RETURN".

(C)

At this point, the algorithm has arrived at two answers; i.e., that 3 and 31 are both integers. The first could not be immediately rejected because the algorithm has no global overview which informs it that there is no syntactic type following integer which would account for the rest of the string. At level zero however, we may eliminate such results, and the single assertion that 31 is indeed an integer remains.

We now consider the problem of left recursion. Suppose one wrote canon 38 in the following manner.

39. $i \text{ integer } \cup d \text{ digit } \mid id \text{ integer }$

The defined set integer has not been altered, but the algorithm will no longer function correctly. Note that whenever the program operates recursively to determine the definition of integer (steps 1, 10, 20), the length of the input string has been reduced by one character. Unless the scan proceeded from right to left, the program using the canon above would be caught in an endless loop, terminated only by the exhaustion of memory. Although it would be possible to devise a scheme to avoid the problem and still interpret the canon correctly, this would require some substantial effort which adds nothing to the scope or generality of this work. Instead, the canons are inspected for left recursion and rejected if it occurs. This constraint does not prevent the definition of any set of strings which could otherwise be defined.

The example brings out one other problem. At different points in the procedure (e.g. steps 42 and 43), the program must handle several possible answers which result from the various ways in which the canons may define the input. On a theoretical level this presents no problem, but in practice the manipulation of multiple large and nearly identical lists may exhaust memory. For this reason, one should follow two suggestions in using the system. Firstly, all syntactic types should be defined in as little

context as possible, so that the legality of a particular string is immediately apparent, and does not depend on a construction occurring much further along in the input. In particular, the canonic system should not allow the input string to be parsed in several different ways, only to discover much later that only one is legal. To do so involves the risk of exhausting memory. Secondly, the canonic system should be unambiguous; that is, a particular string should be generated by only one production or path of application through the canons. Otherwise, both productions will give rise to results. Although the ambiguity could be eliminated by checking for identity among the results at any particular point, the comparisons would be extremely time consuming.

We turn now to an extension of the algorithm for the case in which we wish to consider evaluating a predicate of degree greater than one, for which one or more of the terms are not known and are desired as translated output. The algorithm is presented at an arbitrary recursive level with input of arbitrary degree. For some of the input terms a character string is provided; some are merely marked "needed". Imagine a pointer positioned as before to the left of every term of the input set for which a character string is provided.

1. The program considers in sequence those canons directly defining the input in question, and performs the following steps (2 through 7) for each such canon.
2. The algorithm assembles a list of undefined variables which occur in those terms of the conclusion corresponding to "needed" terms in the input set. These are variables which would not normally be defined during the scan of the conclusion, but for which definitions must be obtained in order to generate the required translations. Variables appearing only in the premises of the canon and not in the conclusion are also added to the list.

3. The input strings provided are matched in sequence against the corresponding terms in the conclusion of the canon. The program skips conclusion terms corresponding to "needed" terms in the input set. If the item in the conclusion at any particular point is a terminal character, the algorithm performs step 4; if a variable, the algorithm performs step 5. When the scan of a term is complete, the program leaves the pointer where it rests and proceeds to the next term for which input is provided. When all such terms are scanned, the algorithm proceeds to step 6.

4. The item in the conclusion is a terminal character. It is compared with the character to the right of the pointer in the input string. If they are identical, the program returns to step 3 with the pointer shifted right one position. If they differ, the scan fails at this point and the algorithm returns to step 1 to pursue alternative definitions for the input.

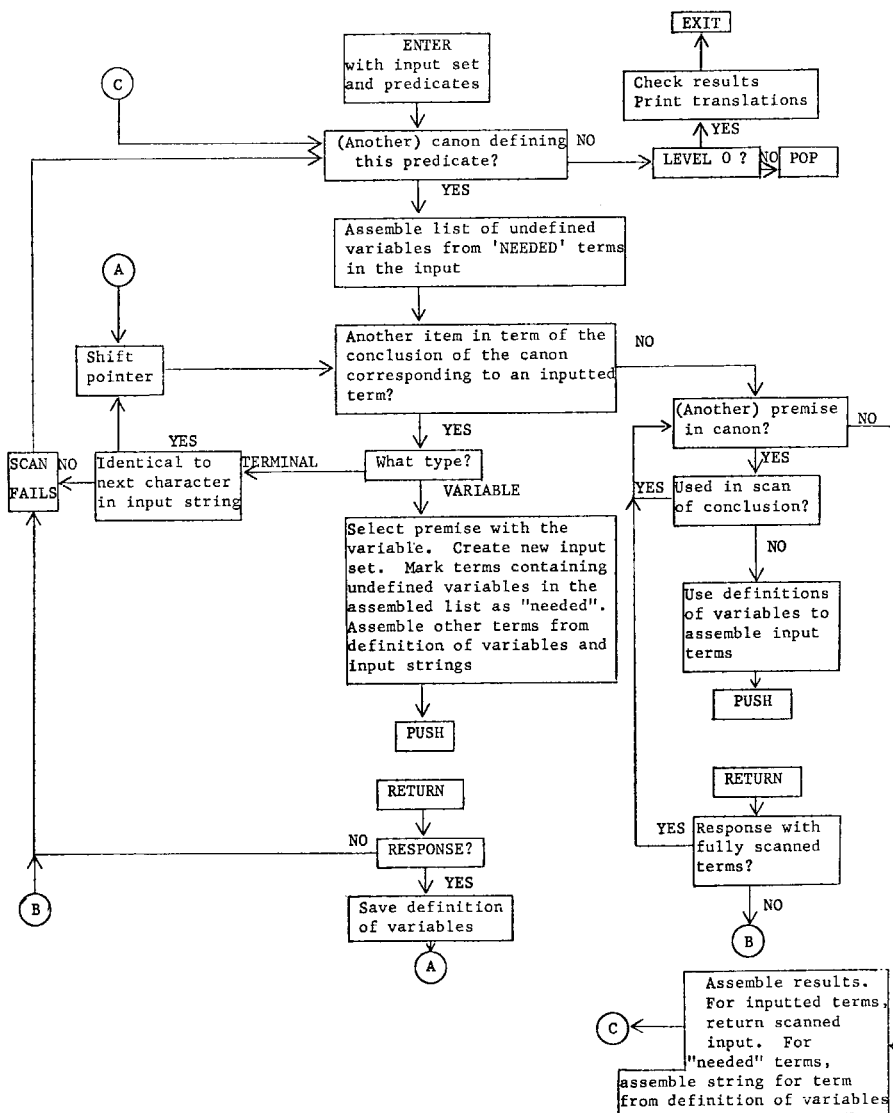
5. The item in the conclusion is a variable, and the algorithm must operate recursively to determine its definition. The program assembles a new input sequence from one of the premises in which the input appears. For the other terms in the premises, it assembles a character string if the variables therein have been defined. If one or more of the variables is undefined and in the "needed" list, it marks the term as "needed". Otherwise, the term is marked as unneeded. The program saves its state and returns to step 1 with the assembled input set for the chosen premise predicate. Upon return, if there is no response, the scan fails. If there is a response, the pointer of the input string is advanced accordingly, the definition of the undefined variables recorded, and the algorithm returns to step 3.

6. The scan of the conclusion is complete. Those premises which were not employed during the scan to generate definitions must now be verified. For these premises, the proper input strings for the terms are assembled

from the now-defined variables, and the algorithm operates recursively to determine whether the premise is satisfied. When all unchecked premises have been satisfied, the algorithm proceeds to the final step. If the return from recursion produces no response, or an input string not fully scanned, the scan fails and the algorithm returns to step 1 to consider any remaining canons.

7. If the scan succeeded, the results for the next higher level of recursion are assembled. For each term given as a string, the string is returned with the mental pointer moved to a position following the last character inspected in the conclusion scan. For each "needed" term, the definition of the term is assembled from the terminal characters and the now-defined variables in that term of the conclusion. If there are more canons to be considered, the algorithm returns to step 1. If not at level 0, the program then pops to the next higher level. If the zero recursion level has been reached, the evaluation is nearly complete. The results are checked to determine if there is a response in which all given terms have been fully scanned. If so, the "needed" terms are outputted. If not, there is a syntax error in the input. The expanded algorithm is presented as a flowchart below.

Flowchart of General Algorithm



A step-by-step example such as the previous table would be unduly lengthy when considering a non-trivial evaluation of a predicate of degree greater than one. Instead, consider as an example the action of the algorithm at the highest level of recursion as it seeks to determine whether an input string is a legal MINI MAD program. The only relevant canon is the last one.

40. $\langle s \langle p \langle r \rangle \text{ program with label lists } \langle r \langle s \rangle \text{ in} \rangle$
 | p END OF PROGRAM * MINI MAD program

The algorithm is presented with an input string which is possibly a member of the set MINI MAD program. Before beginning to scan the input, the program determines that s and r cannot be defined in terms of the input, and places these variables in the undefined list. It then begins the match of the input string against the conclusion of the canon. Since the item in the conclusion is a variable, it turns to the first premise, which contains p as a variable, in order to determine the definition of p in terms of the input string. Since s and r are in the undefined list, it marks these terms as "needed", and operates recursively to determine whether p is valid, and to produce s and r. The algorithm is presented at the next lower level with an ordered triplet in which the first and third elements are "needed" flags, and the second element an input string. If the input is indeed valid, excluding the requirement that all reference labels be defined, the algorithm will scan the input string at progressively deeper levels of recursion, eventually parsing out the statement labels, the various statements, etc. Since the first and third terms of program with label lists are "needed", it will build up these terms from the various statement and reference labels in the program, as directed by the canons which define program with label lists. Eventually, the algorithm will return to level zero. If there is no response returned from the lower level, the scan failed. If there is a response, it will consist of the input

string with the pointer shifted to the right, and the accompanying lists which comprise the first and third terms of program with label lists. The remaining part of the input string is then checked to see whether it consists of END OF PROGRAM*. S and r are now defined. In order to verify the second premise, the algorithm assembles an input set from r and s, and operates recursively to determine if the two lists satisfy the relationship in. Upon return, if there is a response, the program checks to see that both terms are fully scanned; that is, that the definitions of r and s agree in both premises. Since both premises are now satisfied, the algorithm returns the scanned input string as a response. The program is at level 0, and control is given to a final routine which insures that, if there is a response, the input string has been fully scanned. The routine prints out a message to the effect that the input was or was not legal MINI MAD.

We turn now to the problems which may be encountered in evaluating the input in this manner. The potentially most disastrous problem is that of deciding how to generate the definition of variables not defined by the input. In the example above, there is no deterministic way of discovering from the one canon alone why the algorithm should not employ the second premise to generate the label lists. In this case, both terms of an input set would be marked "needed", and the canon would operate recursively to determine the members of the set. The definitions would be inserted, one at a time, into the first premise until the correct ordered pair for the particular input were found. Unfortunately in is an infinite set. Thus, if both terms are marked "needed", the algorithm sets about generating all possible members of the set and speedily exhausts memory. On the other hand, when the definitions for r and s are determined in conjunction with the scan of p as terms in program with label lists, only one ordered pair

of label lists will be produced and inserted in the second premise. A similar but less serious problem might arise in determining the definition of p , if there were more than one premise containing p . Again, the choice of one premise over the other as a vehicle for determining the definition of p might result in a markedly different number of returned responses. These problems have been solved by transferring the decision to the user, who indicates how the definition of a variable should be determined by marking one appearance of the variable in the premises with a prefixed dollar sign. When the program encounters the variable in the conclusion, it will employ the premise in which the variable appears with the dollar sign as the vehicle to determine its definition. If there is no dollar sign, the program uses the premise in which the variable first occurs. Similarly, when considering the other terms of the chosen premise, the algorithm will mark the term as needed only if the variables therein are prefixed with the dollar sign, or if there is no other term in which they appear.

Another simplification is introduced in order to ease the programming effort. The restriction that premise terms contain one and only one variable reduces the complexity of the list manipulation which the program must perform. Again, this does not prevent the definition of sets whose definition is otherwise possible. The premises in the canonic system which defines MINI MAD contain one and only one variable. An important point is that with the restriction we have placed on canonic systems we have in no way diminished their power.

This completes the description of the algorithm on the procedural level. The details of the use of the program, with examples, are described in appendix 3. We now turn to the intriguing question of the practicality of the canonic translator as a useful compiler.

The present program is wholly experimental, and we intend to use it to study the translation process. Three limitations exclude it from serious consideration as a practical device.

1. Speed. The program runs, conservatively, over 1000 times more slowly than a normal compiler.

2. Limitations on input. The program cannot accommodate large quantities of input data.

3. Error indications. If the scan fails, the program pinpoints the last character inspected in the input string, but goes no further. Thus, only one syntax error is detected per compilation.

I feel these limitations can be overcome, and that an implementation of the algorithm might be extremely useful in acting as a trial compiler in the design of a language, or as a regular compiler for lesser used languages where the additional efficiency of a dedicated compiler is not worth the effort necessary to produce one. I shall not consider the use of the algorithm for other language systems, such as the proof of theorems in boolean algebra. The further restrictions imposed on the generality of the algorithm in order to overcome the three limitations will probably reduce its usefulness in other more exotic areas. The proposals follow in order of increasing returns and commensurate restrictions on the algorithm.

1. Redesign and rewrite the program in assembly language. The program as it now stands is the MAD language in neither elegantly designed nor brilliantly executed. The pressure of time and the necessity to have the program work no matter how clumsily, prevented extensive streamlining.

2. Develop, perhaps in conjunction with proposal 1, a list processing system or data structure designed specifically for the algorithm. The SLIP list-processing system is elegantly designed, but its generality necessarily reduces its efficiency for this task. Measure 1 and 2 might provide a five-fold increase in speed, and a doubling of input handling

capacity.

3. Presently, all strings must be members of defined sets in order for premises to be asserted. Consider placing the left of the assertion sign premises which are true if and only if the definition of the variables are not members of the defined sets. Presently, it requires on the order of 26 2/2 canons to define the predicate differ for all letters of the alphabet. By defining a predicate same, as below, one could reduce this to 27 canons.

$$\vdash \langle A \langle A \rangle \rangle_{\Delta} \langle B \langle B \rangle \rangle_{\Delta} \dots \langle Z \langle Z \rangle \rangle_{\Delta}; \text{ same } \langle x \langle y \rangle \rangle_{\sim \text{same}} \vdash \langle x \langle y \rangle \rangle_{\text{differ}}$$

The sign \sim indicates that the ordered pair $x \langle y \rangle$ must not be a member of the set same in order to be a member of the set differ. This procedure would involve problems in originally defining variables, but could be used in premises which would only be verified after the variables have been defined. A moderate increase in speed would result, but the mathematical basis for canonic systems might well be destroyed. The possible implications of such a modification are vast and unexplored.

4. The compilation of a program never produces two different translations. This fact raises questions about the efficiency of handling multiple results at many points in the procedure (e.g. in the example for the simplified algorithm). A program, at any point in the scan of the source statement, is either possibly syntactically valid or definitely invalid. The source statements cannot be construed in several different syntactically valid ways. Consider establishing the rule that the algorithm, at any point in the recursion, returns only the first valid definition it discovers for the predicate. Assume the definition of integer were as follows.

$$41. \text{ d } \underline{\text{digit}} \not\vdash \text{ i } \underline{\text{integer}} \vdash \text{ d i } \underline{\text{integer}}$$

$$42. \text{ d } \underline{\text{digit}} \vdash \text{ d } \underline{\text{integer}}$$

Note that the recursive definition precedes the simpler canon, and the program considers it first. The action of the algorithm will be such that it continually operates recursively, eliminating a digit at each level, until it encounters a character other than a digit. The algorithm then "backs-up" one level, considers the alternative definition, and returns only one answer - an integer of the longest possible length, which is the definition actually desired. The implications of such a restriction are vast. By suitably positioning the non-recursive canons, one immediately eliminates more than half of the searching the program must perform. More importantly, such a rule eliminates all the list manipulation and duplication the program must presently execute. The manipulations are largely responsible for the complexity and inefficiency of the present implementation. Finally, such a restriction eliminates much "back-tracking", and makes it possible to contemplate a single, top-to-bottom pass of the input from auxiliary storage. Likewise, only one set of translation and "needed" lists must be built up, and this makes it possible to arrange the lists in a more conventional and more efficient format. The careful and imaginative implementation of this restriction might improve the speed of compilation by a factor of 50, and make the input capacity of the program comparable to that of conventional compilers. The usefulness of the translator for more general purposes would be, however, severely restricted.

5. One might consider using external subroutines to perform those functions (e.g. in and notin) clumsily handled by an algorithm which must essentially reverse the canonic production of the defined strings. If, as a result of proposal 4, the lists were arranged in a more conventional fashion, such subroutines might be easily implemented.

6. Finally, "system predicates" might be useful. The implementation of the algorithm would consider such elementary predicates as letter, digit

and differ to be understood, so that they need not be defined. Determining that A differs from B by testing whether or not B is one of the other 25 letters is hardly an efficient procedure. Such a provision might greatly speed the compilation.

We have not considered the uses of the algorithm in areas other than language translation, and the implementation of some of these measures, particularly 4, would severely hamper the ability of the algorithm to perform the intent of the canonic system. Other measures, particularly 1 and 6, might still prove useful. I have also avoided proposing a means of dealing with the problem of error indications. This problem might well be the most difficult to solve, but should probably consist of mechanism whereby the algorithm backtracks one syntactic type (e.g. statement) from the one in which the error was detected, skips the syntactic type, and proceeds from there on. Such a procedure might well produce fast and efficient syntax error elimination similar to that produced by a normal compiler.

Canonic systems are extremely powerful mechanisms for the definition of complicated strings. The areas in which canonic systems are applicable, and the possibilities for future study, are both vast and exciting. The possibility of a truly practical generalized compiler implemented through canonic systems deserves further investigation.

Appendix 1.

A.Canonic system specification of the syntax of MINI MAD.

1. Digit \vdash 0 Δ 1 Δ 2 Δ ... Δ 8 Δ 9 digit
2. Integer \vdash d digit \vdash d integer
 \vdash d digit ϕ i integer \vdash di integer
3. Label \vdash A Δ B Δ C label
4. Differ \vdash $\langle A < B \rangle$ Δ $\langle A < C \rangle$ Δ $\langle B < C \rangle$ differ
 \vdash $\langle x < y \rangle$ differ Δ $\langle y < x \rangle$ differ
5. Notin \vdash label \vdash $\langle \Delta \rangle$ notin
 \vdash $\langle x, y \rangle$ notin ϕ $\langle x, l \rangle$ differ \vdash $\langle x < l, y \rangle$ notin
6. In \vdash $\langle \Delta \Delta \rangle$ in
 \vdash $\langle x < y \rangle$ in ϕ Δ label \vdash $\langle x < l, y \rangle$ in
 \vdash $\langle x < y \rangle$ in ϕ label \vdash $\langle l, x < l, y \rangle$ in
 \vdash $\langle x < y \rangle$ in ϕ $\langle z < y \rangle$ in \vdash $\langle xz < y \rangle$ in
7. Variable \vdash X Δ Y Δ Z variable
8. Expression \vdash v variable \vdash v expression
 \vdash i integer \vdash i expression
 \vdash v variable \vdash x expression \vdash v+x expression
 \vdash i integer ϕ x expression \vdash i+x expression
9. Conditional \vdash Δ conditional
 \vdash x expression ϕ y expression \vdash WHENEVER x .E. y, conditional
10. Program with label lists
 \vdash $\langle \Delta \Delta \Delta \rangle$ program with label lists
 \vdash $\langle s < p < r \rangle$ program with label lists ϕ v variable ϕ x expression ϕ
 \vdash c conditional \vdash $\langle s$ CV=X * $p < r \rangle$ program with label lists

$\langle s_< p_< r \rangle$ program with label lists \downarrow $\langle l_< s \rangle$ notin \downarrow
v variable \downarrow x expression \downarrow c conditional \downarrow
 $\langle l_< s_<$ cv=x * p r \rangle program with label lists

$\langle s_< p_< r \rangle$ program with label lists \downarrow m label \downarrow c conditional
 \downarrow $\langle s_<$ c TRANSFER TO m * p_< m, r \rangle
program with label lists

$\langle s_< p_< r \rangle$ program with label lists \downarrow m label \downarrow $\langle l_< s \rangle$ notin
 \downarrow c conditional \downarrow $\langle l_< s_< l$ c TRANSFER TO m * p_< m, s \rangle
program with label lists

11. MINI MAD program

$\langle s_< p_< r \rangle$ program with label lists \downarrow $\langle r_< s \rangle$ in
 \downarrow p END OF PROGRAM * MINI MAD program

*This empty page was substituted for a
blank page in the original document.*

B. Canonic system specification of the syntax and translation of MINI MAD into PSEUDO FAP. The dollar sign in PSEUDO FAP indicates "this location".

1. Digit \vdash $0 \Delta 1 \Delta 2 \Delta \dots \Delta 8 \Delta 9$ digit
2. Integer \vdash d digit \vdash d integer
 \vdash d digit \downarrow i integer \vdash di integer
3. Label \vdash $A \Delta B \Delta C$ label
4. Differ \vdash $\langle A \Delta B \rangle \Delta \langle A \Delta C \rangle \Delta \langle B \Delta C \rangle$ differ
 \vdash $\langle x \Delta y \rangle$ differ \vdash $\langle y \Delta x \rangle$ differ
5. Notin \vdash ℓ label \vdash $\langle \ell \Delta \Lambda \rangle$ notin
 \vdash $\langle x \Delta y \rangle$ notin \downarrow $\langle x \Delta \ell \rangle$ differ \vdash $\langle x \Delta \ell, y \rangle$ notin
6. In \vdash $\langle \Lambda \Delta \Lambda \rangle$ in
 \vdash $\langle x \Delta y \rangle$ in \downarrow ℓ label \vdash $\langle x \Delta \ell, y \rangle$ in
 \vdash $\langle x \Delta y \rangle$ in \downarrow ℓ label \vdash $\langle \ell \Delta x \Delta \ell, y \rangle$ in
 \vdash $\langle x \Delta y \rangle$ in \downarrow $\langle x \Delta y \rangle$ in \vdash $\langle xz \Delta y \rangle$ in
7. Variable \vdash $X \Delta Y \Delta Z$ variable
8. Expression \vdash v variable \vdash $v \langle \text{CLA } v^* \rangle$ expression
 \vdash i integer \vdash $i \langle \text{CLA } =i^* \rangle$ expression
 \vdash v variable \downarrow $\langle x \Delta y \rangle$ expression \vdash $\langle v+x \Delta y$ ADD $v^* \rangle$ expression
 \vdash i integer \downarrow $\langle x \Delta y \rangle$ expression \vdash $\langle i+x \Delta y$ ADD $=i^* \rangle$ expression
9. Conditional \vdash $\langle \Lambda \Delta \Lambda \rangle$ conditional
 \vdash $\langle x \Delta y \rangle$ expression \downarrow $\langle u \Delta v \rangle$ expression
 \vdash \langle WHENEVER $x.E. u, \langle y$ STO TMP^*
 $\text{SUB } \text{TMP}^*$ $\text{TNZ } \$+3 \rangle$ conditional
10. Program with translation
 \vdash $\langle \Lambda \Delta \Lambda \Delta \Lambda \Delta \Lambda \rangle$ program with translation

$\langle s_c p_c r_c t \rangle$ program with translation \downarrow v variable
 $\downarrow \langle x_c y \rangle$ expression $\downarrow \langle c_c d \rangle$ conditional \downarrow
 $\langle s_c \quad cv=x * p_c r_c \quad y \quad STO TNP * \quad$
 $d \quad CIA TNP * \quad STO V * t \rangle$
program with translation

$\langle s_c p_c r_c t \rangle$ program with translation $\downarrow \langle l_c s \rangle$ notin \downarrow
v variable $\downarrow \langle x_c y \rangle$ expression $\downarrow \langle c_c d \rangle$ conditional
 $\downarrow \langle l_c, s_c l_c \quad cv=x * p_c r_c \quad$
 $f y \quad STO TNP * \quad d \quad CIA TNP * \quad STO V * t \rangle$

$\langle s_c p_c r_c t \rangle$ program with translation $\downarrow m$ label $\downarrow \langle c_c d \rangle$ conditional \downarrow
 $\langle s_c \quad c \quad TRANSFER TO m * p_c m, r_c \quad d \quad TRA m * \quad NOP * t \rangle$
program with translation

$\langle s_c p_c r_c t \rangle$ program with translation $\downarrow m$ label $\downarrow \langle l_c s \rangle$ notin $\downarrow c$ conditional
 $\downarrow \langle l_c, s_c l_c \quad c \quad TRANSFER TO m * p_c m, s_c l_c \quad d \quad TRA m * \quad NOP * t \rangle$
program with translation

11. MINI MAD - PSEUDO FAP

$\langle s_c p_c r_c t \rangle$ program with translation $\downarrow \langle r_c s \rangle$ in
 $\downarrow \langle p \quad END OF PROGRAM * \quad t \quad HLT * \quad$
 $TMP DEC * \quad TNP DEC * \quad END * \rangle$
MINI MAD - PSEUDO FAP

As an example, the program given previously in the text is reproduced below with the equivalent PSEUDO FAP program.

```
A   X = 15
B   X = X+1
    WHENEVER X .E. 123, TRANSFER TO A
    TRANSFER TO B

A   CLA =15
    STO X
B   CLA =1
    ADD X
    STO X
    CLA X
    STO TNP
    CLA =123
    SUB TNP
    TNZ $ + 3
    TRA A
    NOP
    TRA E
TMP DEC
TNP DEC
    HLT
    END
```

Appendix 2.

A canonic system specification for the syntax of SNOBOL.

The canonic system presented in this appendix defines the syntax of SNOBOL as implemented on the 7094 CTSS system at MIT. The language is used for string processing and contains statements for string matching, replacing, deleting and inserting. The language also has a few arithmetic capabilities. Those not familiar with the language may find reference 5 useful.

The canonic system is listed below. λ represents a space.

1. $\{ A_{\Delta} B_{\Delta} C_{\Delta} \dots X_{\Delta} Y_{\Delta} Z_{\Delta} \}$ letter
2. $\{ 0_{\Delta} 1_{\Delta} 2_{\Delta} \dots 7_{\Delta} 8_{\Delta} 9_{\Delta} \}$ digit
3. x letter ϕ digit $\{ x_{\Delta} y_{\Delta} .$ name character
4. x name character $\{ x_{\Delta} , ' _ { \Delta } * + - / ? = \$$ label character
5. x name character $\{ x_{\Delta} , () _ { \Delta } * + - / ? = \$$ string character
6. x string character $\{ x_{\Delta} '$ character
7. $\{ + _ { \Delta } - / _ { \Delta } *$ operator
8. $\{ \downarrow$ tab
9. $\{ \downarrow)$ carriage return
10. x spaces $\{ \lambda _ { \Delta } x \lambda$ spaces
11. $a_{\Delta} b_{\Delta} c_{\Delta} d_{\Delta} e_{\Delta} f_{\Delta}$ name character $\{ a_{\Delta} ab_{\Delta} abc_{\Delta} abcd_{\Delta} abcde_{\Delta} abcdef_{\Delta}$ string name
12. x string character ϕ string $\{ xy$ string
13. x string $\{ 'x'$ literal
14. x letter ϕ digit ϕ z label character ϕ a label $\{ x_{\Delta} y_{\Delta} az_{\Delta}$ label
15. x digit ϕ y integer $\{ x_{\Delta} yx_{\Delta}$ integer

16. x string name \downarrow literal \downarrow x_{Δ} operand
17. x_{Δ} operand \downarrow expression \downarrow operator \downarrow spaces
 \downarrow $xsvsy_{\Delta}$ $xsvsz_{\Delta}$ $zsvsy_{\Delta}$ (z) expression
18. x operand \downarrow expression \downarrow term \downarrow spaces
 \downarrow x_{Δ} y_{Δ} zsx_{Δ} zsy term
19. x term \downarrow \wedge_{Δ} concatenation
20. x string name \downarrow $*x*$ variable name
21. x_{Δ} string name \downarrow integer \downarrow $*x/y*$ $*x/'z'*$ fixed length name
22. x string name \downarrow $*(x)*$ balanced name
23. x string name \downarrow literal \downarrow u_{Δ} term \downarrow w indirect name
 \downarrow spaces \downarrow $\$x_{\Delta}$ $\$y_{\Delta}$ $\$(w)_{\Delta}$ $\$(usw)_{\Delta}$ $\$(wsu)_{\Delta}$ $\$(uswsv)$ indirect name
24. $\langle A_{\Delta} B \rangle_{\Delta} \langle A_{\Delta} C \rangle_{\Delta} \dots \langle B_{\Delta} C \rangle_{\Delta} \dots \langle =_{\Delta} \rangle_{\Delta}$ differ
25. $a_{\Delta} e_{\Delta} f$ string \downarrow $\langle b_{\Delta} c \rangle$ differ \downarrow $\langle ace_{\Delta} abf \rangle$ different
26. $\langle x_{\Delta} y \rangle$ different \downarrow $\langle y_{\Delta} x \rangle$ different
27. x label \downarrow y list \downarrow \wedge_{Δ} y_{xw} list
28. x list \downarrow $\langle \wedge_{\Delta} x \rangle$ in
29. x_{Δ} list \downarrow $\langle w_{\Delta} xy \rangle$ in \downarrow l label \downarrow $\langle wlw_{\Delta} xlwy \rangle$ in
30. $\langle w_{\Delta} xy \rangle$ in \downarrow $\langle u_{\Delta} xy \rangle$ in \downarrow $\langle wu_{\Delta} xy \rangle$ in
31. x label \downarrow $\langle x_{\Delta} \wedge \rangle$ notin
32. x_{Δ} label \downarrow $\langle x_{\Delta} y \rangle$ different \downarrow $\langle x_{\Delta} z \rangle$ notin \downarrow $\langle x_{\Delta} zy \rangle$ notin
33. x string name \downarrow concatenation \downarrow spaces
 \downarrow $xs=sy$ assignment statement
34. x operand \downarrow expression \downarrow u variable name \downarrow v fixed length name
 \downarrow w balanced name \downarrow z indirect name \downarrow u_{Δ} v_{Δ} w_{Δ} x_{Δ} y_{Δ} z
scan operand

35. x scan operand \downarrow z scan \downarrow s spaces \downarrow x Δ xsz scan
36. x operand \downarrow y concatenation \downarrow z scan \downarrow s spaces
 \downarrow xsz Δ $xszs=sy$ scan statement
37. \downarrow -EJECT Δ -LIST Δ -NULOP OP Δ -PCC Δ -SPACE Δ -TITLE
 Δ -UNLIST control word
38. x operand \downarrow a arguments \downarrow Λ Δ x Δ a , x Δ x , a Δ 'a' Δ (a) arguments
39. x string name \downarrow a arguments \downarrow $x(a)$ string name \downarrow
 $x(a)$ system function
40. x label \downarrow y indirect name \downarrow $\langle x, xw \rangle \Delta \langle y, \Lambda \rangle$ reference label
41. $\langle x, y \rangle \Delta \langle w, z \rangle$ reference label \downarrow $\langle / (x), y \rangle \Delta \langle / S(x), y \rangle \Delta \langle / F(x), y \rangle$
 $\Delta \langle / S(x)F(w), yz \rangle$ branch
42. x scan statement \downarrow y assignment statement \downarrow z system function
 $\langle u, v \rangle$ branch \downarrow s spaces \downarrow $\langle x, \Lambda \rangle \Delta \langle y, \Lambda \rangle \Delta \langle xsu, v \rangle$
 $\Delta \langle ysu, v \rangle \Delta \langle z, \Lambda \rangle \Delta \langle zsu, v \rangle$ right hand side
43. $\langle x, y \rangle$ right hand side \downarrow $\langle \text{END} \downarrow x, y \rangle \Delta \langle \text{END}, \Lambda \rangle$ end card
44. \downarrow $\langle \Lambda, \Lambda, \Lambda \rangle$ program string
45. $\langle p, q, r \rangle$ program string \downarrow x control word \downarrow $\langle p, qx, \downarrow, r \rangle$ program string
46. $\langle p, q, r \rangle$ program string \downarrow $\langle x, y \rangle$ right hand side \downarrow u label
 \downarrow $\langle u, p \rangle$ notin \downarrow $\langle p, q \downarrow x, \downarrow, ry \rangle \Delta \langle puw, qu \downarrow x, \downarrow, ry \rangle$ program string
47. $\langle p, q, r \rangle$ program string \downarrow $\langle \text{END}, p \rangle$ notin \downarrow $\langle x, y \rangle$ end card
 \downarrow $\langle ry, p \rangle$ in \downarrow qx program

Appendix 3.

Use of Program.

The program which implements the algorithm allows the user to type in a series of canons defining a set of strings, followed by the input he wishes to have analyzed. The program then scans the input string or strings for correct syntax. If the input is syntactically correct, a message to this effect is printed. Further, if the input is defined as only one of several terms in the final predicate of the canonic system, the other terms corresponding to the input may be produced. If the scan fails, the program identifies the character in the input string which was the last character inspected.

The sequence of messages and the proper responses as the program executes on the MIT CTSS system are as follows.

INPUT CANONS.

A set of canons may now be input, subject to the restrictions described in the text and summarized briefly below.

1. Canons may contain only one conclusion.
2. The terms of the premise predicates may contain one and only one variable, and no terminal characters.
3. Left recursion in all terms of a predicate is not permitted. Partial left recursion evokes a warning message.

The user inputs the canons according to the following rules which implement the punctuation of the canonic system.

1. Strings of terminal characters must be enclosed in break characters

(' / or *).

2. The digit 1 and the digit 2 when not enclosed in breaks represent respectively a tab and a carriage return.
3. Letters represent variables. All variables used in the conclusion must be defined in the premises.
4. Predicate names must consist of six characters or less, and be enclosed in hyphens.
5. The terms of a predicate are separated by periods.
6. The premise remarks of a canon are separated by commas.
7. An equals sign replaces the assertion sign.
8. Spaces and carriage returns are ignored except when enclosed in breaks, but each line may not contain more than one canon.

The examples at the end of this appendix will serve to clarify the syntax rules.

After the last canon, the user types 'end' at the beginning of a line. The program responds with the following sequence after checking that all predicates used as premises are defined as conclusions.

CONSISTENT SET OF CANONS.

LIST OF DEFINED PREDICATES AND DEGREEES.

The predicates typed in are then listed in the order in which they first appeared.

INPUT OF SOURCE STRINGS.

TYPE FINAL PREDICATE.

The user responds by typing the predicate name which defines the input string he wishes the program to consider.

TYPE -NONEED-, -NEED- OR -INPUT- FOR EACH TERM.

TERM NUMBER n OF -predicate-

At this point the user declares which terms of the final predicate he wishes to input and which terms he desires as translation. 'Noneed' indicates that he wishes neither to input the term nor receive it as output. 'Need' indicates he wishes to receive the term as a translation. 'Input' means that he wishes to type in an input string for the term. In this case, the program responds.

INPUT STRING. EXTRA CARRIAGE RETURN INDICATES END.

The user may now type in input which will be verified for syntactic correctness, and for which the program will produce output corresponding to 'needed' terms. Carriage returns are counted as characters. If the user wishes instead to input card images, he may do so by typing in 80 characters or more. The input is truncated at 80 characters and in this case the carriage return will not be counted.

After all terms of the final predicate have been considered, the program types this message.

TYPE 0, 1 OR 2 FOR DEPTH OF COMMENTS.

The user responds by typing a single number. If 0, the program will print only the final results. If 1, it will remark on extraordinary conditions which occur. Typing 2 results in messages whenever the program "pops" or "pushes". A larger number will result in the output of various lists which comprise the intermediate results of the scan. These lists, while useful during program

during program debugging, are rather incomprehensible except to those familiar with both the program and the SLIP system.

The program then types

SCAN BEGINS.

When the program returns to the zero level of recursion, it will type out the results of the analysis. If the scan succeeds, and if terms are 'needed', these terms are printed. If there is more than one translation, all will be printed. In the examples which follow, the execution time, which is printed in seconds at the end of the run, indicates the problems of execution speed to be overcome if one wishes to make a practical canonic translator.

There are three examples of canonic translation. The first is relatively simple. It illustrates a scheme for coding messages by replacing the letters in the message with their successors in the alphabet. The second example demonstrates the construction of an expression in MINI MAD and the corresponding PSEUDO FAP instructions. The third example, an extension of the second, demonstrates the construction of an assignment statement in MINI MAD and the translation into PSEUDO FAP. Note that no data cells were reserved, although this could have been easily implemented. A final example illustrates the error analysis of the program.

resume thesis
W 2121.4
INPUT CANONS.

= 'a'. 'b' -pair-
= 'b'. 'c' -pair-
= 'c'. 'd' -pair-
= 'd'. 'e' -pair-
= 'e'. 'f' -pair-
= 'f'. 'a' -pair-
x.y -pair- = x.y -code-
x.y -pair-, u.v -code- = xu.yv -code-
u.v -code- = u2 . v' is the coded message for 'u2 -messag-
end

CONSISTENT SET OF CANONS.

LIST OF DEFINED PREDICATES AND DEGREES.

1. - PAIR- 2
2. - CODE- 2
3. -MESSAG- 2

INPUT OF SOURCE STRINGS.
TYPE FINAL PREDICATE.
messag

TYPE -NONEED-, -NEED- OR -INPUT- FOR EACH TERM.

TERM NUMBER 1 OF -MESSAG-
input

INPUT STRING. EXTRA CARRIAGE RETURN INDICATES END.

abcdef

TERM NUMBER 2 OF -MESSAG-
need

TYPE 0, 1 OR 2 FOR DEPTH OF COMMENTS.
0

SCAN BEGINS.

SCAN SUCCESSFUL.
TRANSLATED OUTPUT (IF ANY) FOLLOWS.

TERM NUMBER 2.

BCDEFA IS THE CODED MESSAGE FOR ABCDEF

END OF RUN.
EXITM CALLED. GOODBYE.
R 7.150+6.983

resume thesis
w 2127.5
INPUT CANONS.

```
= 'x' -variab-  
= 'y' -variab-  
= 'z' -variab-  
= '1' -digit-  
= '2' -digit-  
= '3' -digit-  
d -digit- = d -integ-  
d -digit-, i -integ- = di -integ-  
i -integ- = i . '   cla = 'i2 -expres-  
v -variab- = v . '   cla 'v2 -expres-  
i -integ-, x.y -expres- = i+'x . y'   add = 'i2 -expres-  
WARNING- PARTIAL LEFT RECURSION IN LINE NUMBER 11  
v -variab-, x.y -expres- = v+'x . y'   add 'v2 -expres-  
WARNING- PARTIAL LEFT RECURSION IN LINE NUMBER 12  
x.y -expres- = x2 . 'this is the translation for. 'x22  
   y'   end'2 -exampl-  
end
```

CONSISTENT SET OF CANONS.

LIST OF DEFINED PREDICATES AND DEGREES.

- 1. -VARIAB- 1
- 2. - DIGIT- 1
- 3. - INTEG- 1
- 4. -EXPRES- 2
- 5. -EXAMPL- 2

INPUT OF SOURCE STRINGS.

TYPE FINAL PREDICATE.

exampl

TYPE -NONEED-, -NEED- OR -INPUT- FOR EACH TERM.

TERM NUMBER 1 OF -EXAMPL-

Input

INPUT STRING. EXTRA CARRIAGE RETURN INDICATES END.

x+123+y+321+z

TERM NUMBER 2 OF -EXAMPL-

need

TYPE 0, 1 OR 2 FOR DEPTH OF COMMENTS.

0

-51-

SCAN BEGINS.

SCAN SUCCESSFUL.
TRANSLATED OUTPUT (IF ANY) FOLLOWS.

TERM NUMBER 2.

THIS IS THE TRANSLATION FOR $X+12Y+Y+521+Z$

```
CLA Z
ADD #521
ADD Y
ADD #125
ADD X
END
```

END OF RUN.
EXIT: CALLED. GOODBYE.
R 16,755+8.966

```
resume thesis
w 2150.5
INPUT CANONS.
```

```
= 'x' -variab-
= 'y' -variab-
= 'z' -variab-
= '1' -digit-
= '2' -digit-
= '3' -digit-
d -digit- = d -integ-
d -digit-, i -integ- = di -integ-
i -integ- = i . '   cla = 'i2 -expres-
v -variab- = v . '   cla 'v2 -expres-
i -integ-, x.y -expres- = i+'x . y'   add = 'i2 -expres-
WARNING- PARTIAL LEFT RECURSION IN LINE NUMBER 11
v -variab-, x.y -expres- = v+'x . y'   add 'v2 -expres-
WARNING- PARTIAL LEFT RECURSION IN LINE NUMBER 12
v -variab-, x.y -expres- = v='x . y'   sto 'v2 -assign-
x.y -assign- = x2 . 'this is the translation for 'x2
   y'   end'2 -exampl-
end
```

```
CONSISTENT SET OF CANONS.
```

```
LIST OF DEFINED PREDICATES AND DEGREES.
```

1. -VARIAB- 1
2. - DIGIT- 1
3. - INTEG- 1
4. -EXPRES- 2
5. -ASSIGN- 2
6. -EXAMPL- 2

```
INPUT OF SOURCE STRINGS.
TYPE FINAL PREDICATE.
exampl
```

-53-

TYPE -NONEED-, -NEED- OR -INPUT- FOR EACH TERM.

TERM NUMBER 1 OF -EXAMPL-
Input

INPUT STRING. EXTRA CARRIAGE RETURN INDICATES END.

y=x+123+y+3211+z

TERM NUMBER 2 OF -EXAMPL-
need

TYPE 0, 1 OR 2 FOR DEPTH OF COMMENTS.
0

SCAN BEGINS.

SCAN SUCCESSFUL.
TRANSLATED OUTPUT (IF ANY) FOLLOWS.

TERM NUMBER 2.

THIS IS THE TRANSLATION FOR Y=X+123+Y+3211+Z

```
CLA Z
ADD =3211
ADD Y
ADD =123
ADD X
STO Y
END
```

END OF RUN.
EXITM CALLED. GOODBYE.
R 18.866+8.400

-54-

resume thesis
W 2200.4
INPUT CANONS.

= 'this is a test sentence'² -exampl-
end

CONSISTENT SET OF CANONS.

LIST OF DEFINED PREDICATES AND DEGREES.

1. -EXAMPL- i

INPUT OF SOURCE STRINGS.
TYPE FINAL PREDICATE.
exampl

TYPE -NONEED-, -NEED- OR -INPUT- FOR EACH TERM.

TERM NUMBER 1 OF -EXAMPL-
input

INPUT STRING. EXTRA CARRIAGE RETURN INDICATES END.

this is not a test sentence

TYPE 0, 1 OR 2 FOR DEPTH OF COMMENTS.
0

SCAN BEGINS.

SCAN FAILED. SYNTAX ERROR IN INPUT STRING(S).
NO TRANSLATED OUTPUT.

LAST CHARACTER INSPECTED IN TERM 1 WAS N IN MIDST OF FOLLOWING CONTEXT

THIS IS NOT A TEST SENTE

END OF RUN.
EXITM CALLED. GOODBYE.
R .583+2.766

Appendix 4.

Program Listing.

The program listing for the program which implements the canonic translation algorithm is contained in this appendix. The program may be divided into three parts: a preliminary phase which verifies the syntax of the canons typed in and assembles them into a SLIP list structure, the recursive scanning routine which forms the major part of the code, and a final routine which inspects and prints the results. Understanding the code requires a thorough comprehension of the SLIP system developed by Weizenbaum (7). The lack of elegance in the program is quite the fault of the author.

The following table identifying the major parts of the code may prove useful.

<u>Label</u>	<u>Lines</u>	<u>Purpose of Code</u>
NEWORD	57-74	Inputs a line from the typewriter, feeds characters one at a time to the canon-analyzing routine.
	107-285	Reads predicate names and makes various checks (left recursion, degree same as before, etc.) and assembles into list structure.
	383-395	Identifies next variable to be encountered should be marked as the one to use if the variable is needed in the later phase.
	396-434	Inputs variable and assembles into SLIP structure.
EVAL	444-472	Checks that all variables are defined.
PUTIN	505-574	Assembles list structure for input to scan program at zero level.

LUP000 592 Beginning of recursive routine. It is to this point
that the program returns when "pushing".

603-617 Makes an "object time" check for left recursion.

OUTCHK 621-671 Creates the 'needed' list of variables for which
definitions must be found from other than the input
string.

LUP008 677-697 Handles multiple results, each of which must
be analyzed in turn.

717-855 Compares input string with conclusion, "pushing"
to find definition of variables if necessary.

PUSHIT 871-893 Saves state of program and returns to the beginning
of the scan routine.

PRMCHK 897-977 Checks premises of a canon.

ASSMBL 981-1038 Assembles results of scan for next higher level
before "popping".

POP 1042-1075 Uncovers the state of the program and goes to
appropriate return routine.

POP1 1080-1164 Analyzes return resulting from "push" during
scan of conclusion.

POP2 1165-1196 Analyzes return resulting from "push" during
check of premises.

THKGOD 1200-1344 Outputs results of scan.

HERAUS 1348-1350 Exit.

1355-1381 Obtains character from input data area, using
pointer furnished by caller.

1285-1334: 100% answers using input routine at top.

PRTLIST = 1.62: 100% subroutine to print out the content and structure of lists
for debugging.

```

*****
----- M5364      5163      THESIS      MAD FOR  M5364      5163      0
R
R
R
R CANONIC TRANSLATION PROGRAM.
R THIS PROGRAM EMPLOYS A CANONIC SYSTEM AS A BASIS
R FOR RECOGNIZING DEFINED SETS OF STRINGS. THE
R FIRST PART OF THE PROGRAM VERIFIES THE SYNTAX OF
R THE CANONIC SYSTEM WHICH IS INPUT AND
R ASSEMBLES IT INTO A SLIP LIST STRUCTURE.
R THE SECOND PART OF THE PROGRAM OPERATES RECURSIVELY
R TO DETERMINE WHETHER A GIVEN STRING IS A MEMBER
R OF A DEFINED SET OF STRINGS, OR WHETHER THE GIVEN
R STRING IS ONE ELEMENT OF AN ORDERED N-TUPLE
R WHICH IS A MEMBER OF A DEFINED SET. IN THE LATTER
R CASE, THE OTHER ELEMENTS OF THE N-TUPLE MAY BE
R OUTPUT AS TRANSLATIONS.
R
R
R
R N'S INTEGER
R BCOLEAN EQUAL, EOLIND, STRTND, ERRS, NOTIN, IN, SOMERC, ALLRC
R 1 ,LEMPY, INP, DOLIND
R DIMENSION BUFFER (14), INPUT (1000)
R
R INITIALIZATION OF SLIP SYSTEM.
R
R
R INITAS. (0)
R BEGIN  LIST. (SYSTEM)
R        LIST. (NAMES)
R        LINE = 0
R        DEFNUM = 0
R
R
R INITIALIZATION OF SYSTEM
R
R PRINT COMMENT $INPUT CANONS.$
R PRINT COMMENT $ $
R T'O SKIP
R
R INITIALIZATION PRIOR TO READING A CANON
R
R LOOP1  IRALST. (VAR)
R        IRALST. (SVEVAR)
R SKIP   DEF = LIST. (9)
R        LIST. (VAR)
R        LIST. (SVEVAR)
R        NEED = LIST. (9)
R        DCLIND = 0B
R        CCND = 1
R        EQUAL = 0B
R        ECLIND = 1B
R NEWPRM PREM = LIST. (9)
R        TRMNUM = 0
R NEWTRM TERM = LIST. (9)
R
R
R INPUT OF CANONS, CHARACTER BY CHARACTER, RIGHT
R ADJUSTED IN WORD
R
R NEWORD W'R .NOT. ECLIND, T'O GETW
R ROLINE NUMB = RDFLXC. (BUFFER, 84)

```

```

LINE = LINE + 1
ECLIND = 0B
PCS = 6
I = -1
STRTND = 1B
GETW W'R POS .GE. 6
      POS = 0
      I = I + 1
      WORT = BUFFER (I)
E'L
W'R NUMB .E. 0, T'O RDLIN
NUMB = NUMB - 1
W'R NUMB .E. 0, EDLIND = 1B
PCS = POS + 1
WCRD = WORT .RS. 30 .V. $    0$
WCRT = WORT .LS. 6
R
RCHECK TO SEE IF READING ANSWER TO QUESTIONS.
R
W'R COND .E. 8
      T'O RDANS
R
RCHECK FOR REMARK AND END CARDS
R
C'R STRTND
      STRTND = 0B
      W'R WORD .E. $    *$, T'O RDLIN
      W'R BUFFER (0) .A. 77777700000K .E. $END000$
      W'R COND .E. 1, T'O EVAL
      PRNTP. (ERR1)
      V'S ERRI = $LAST CANON IS INCOMPLETE$, 37777777777
IK
      T'O ERRIN
E'L
E'L
R
RCHECK TO SEE IF READING 'LITERAL' OF TERMINAL CHARACTERS
R
W'R COND .E. 6
      W'R WORD .E. BREAK
      COND = 2
O'E
      NEWBOT. (WORD, TERM)
      W'R EDLIND, NEWBOT. (606060606055K, TERM)
E'L
T'O NEWORD
O'R WORD .E. $ $
      T'O NEWORD
R
RCHECK TO SEE IF READING PREDICATE
R
C'R COND .E. 7
      W'R WORD .NE. $    -$
      LENGTH = LENGTH + 1
      W'R LENGTH .E. 7
      PRNTP. (ERR2)
      V'S ERR2 = $TOO MANY CHARACTERS IN PREDICATES
1, 377777777777K
      T'O ERRIN
E'L
R

```

```

RBUILD UP PREDICATE, CHARACTER BY CHARACTER
R
      NAME = NAME .LS. 6 .V. WORD .A. 00000000077K
      T'O NEWORD
R
RSAVE PREDICATE NAME JUST READ IN
R
      O'E
      EQUIV = ITSVAL. (NAME, NAMES)
      W'R EQUIV .E. 0
      DEFNUM = DEFNUM + 1
      EQUIV = DEFNUM .V. TRMNUM .LS. 18
      NEWVAL. (NAME, EQUIV, NAMES)
      E'L
      CHKNUM = EQUIV .RS. 18
      EQUIV = EQUIV .A. 777777K
R
RCHECK DEGREE OF PREDICATE
R
      W'R TRMNUM .NE. CHKNUM
      P'T ERR15, NAME, LINE
      V'S ERR15 = $H'DEGREE OF PREDICATE -1,C6,
1H'- IN LINE NUMBER',I3,H' NOT AS PREVIOUSLY DEFINED'*$
      T'O ZAPALL
      E'L
R
RIF CONCLUSION, MAKE VARIOUS CHECKS AND ADD CANON TO SYSTEM
R
      W'R EQUAL
R
RCHECK FOR LEFT RECURSION
R
      LRECUR = SEQRD. (DEF)
      CHKPRM = SEQLR. (LRECUR, F)
      W'R F .G. 0, T'O CHKILL
      PRMPRM = TOP. (LSTNAM. (CHKPRM))
      1 .A. 777777K
      W'R PRMPRM .NE. EQUIV, T'O LOOP3
      CHECKC = SEQRD. (PREM)
      CHECKP = SEQRD. (CHKPRM)
      SOMERC = 0B
      ALLRC = 1B
      LOOP4
      TERMP = SEQLR. (CHECKP, F)
      TERM2 = SEQLR. (CHECKC, G)
      W'R F .G. 0 .OR. G .G. 0
      W'R SOMERC .AND. ALLRC
      PRNTP. (ERR11)
      V'S ERR11 = $COMPLETE LEFT RECURSION
      1$, 37777777777K
      T'O ERRIN
      O'R SOMERC
      P'T ERR12, LINE
      V'S ERR12 = $H'WARNING- PARTIAL LEFT
      1 RECURSION IN LINE NUMBER',I3*$
      E'L
      T'O LOOP3
      E'L
      TERMP = SEQRD. (TERMP)
      TERM2 = SEQRD. (TERM2)
      LOOP5
      PPPREM = SEQLR. (TERM1, F)
      CONCLU = SEQLR. (TERM2, G)

```

```

W'R F .G. 0 .DR. G .G. 0, T'O LOOP4
W'R F .L. 0 .AND. G .L. 0
W'R PPPREM .E. CONCLU, T'O LOOP5
O'R F .E. 0 .AND. G .E. 0
W'R TOP. (LSTNAM. (CONCLU)) .E.
1 CHKPRM .AND. PPPREM .E. CONCLU
SOMERC = 1B
T'O LOOP4
E'L
E'L
ALLRC = 0B
T'O LOOP4
R
RCHECK FOR ILLEGAL VARIABLE CONSTRUCTION, I.E.
RA PREMISE TERM WITH MORE THAN A VARIABLE.
R
CHKILL READP = SEQRDR. (DEF)
ADVANP PREMP = SEQLR. (READP, F)
W'R F .G. 0, T'O NOUSE
READT = SEQRDR. (PREMP)
ADVANT TERMP = SEQLR. (READT, F)
W'R F .G. 0, T'O ADVANP
NOTIN = 0B
IN = 0B
VARCNT = 0
READV = SEQRDR. (TERMP)
ADVANV VARIAB = SEQLR. (READV, F)
W'R F .G. 0
GDOF01 W'R VARCNT .G. 1
PRNTP. (ERR16)
1, 37777777777K V'S ERR16 = $VARIABLE NOT ISOLATED$
T'O ERRIN
O'R VARCNT .L. 1
PRNTP. (ERR22)
1, 37777777777K V'S ERR22 = $TERM WITHOUT VARIABLES$
T'O ERRIN
E'L
T'O ADVANT.
E'L
W'R F .L. 0, T'O GDOF01
VARIAB = TOP. (VARIAB)
VARCNT = VARCNT + 1
LOOP6 READC = SEQRDR. (SVEVAR)
TERMV = SEQLR. (READC, F)
W'R F .G. 0
NOTIN = 1B
READL = SEQRDR. (NEED)
ADVANL TERML = SEQLR. (READL, F)
W'R F .G. J
NEWBOT. (VARIAB, NEED)
P'T NOTTY, VARIAB
V'S NOTTY = $H'NEED ',C6*$
T'O ADVANV
O'R VARIAB .A. 77777777777K .E. TERML .A.
1 77777777777K SUBST. (VARIAB .V. 77K10, SEQPTR.
I (READL))
T'O ADVANV
E'L

```

```

..... T'D ADVANL
..... O'R TERMV .E. VARIAB
..... IN = 18
..... T'D ADVANV
..... E'L
..... T'D LOOP6
R
RCHECK FOR UNUSED VARIABLE (ONE WHICH OCCURS ONLY ONCE
RIN CANON)
R
NOUSE
ADVANN
..... READL = SEQDR. (NEED)
..... TERM = SEQLR. (READL, F)
..... W'R F .G. D, T'D ADDCAN
..... W'R TERM .A. 77K10 .E. 77K10
..... TERM = TERM .A. 60777777777K
..... T'D ADVANN
..... E'L
..... P'T ERR17, TERM, LINE
..... V'S ERR17 = $H'WARNING- VARIABLE ',RC1,
..... IH' IN LINE NUMBER',13,H' UNUSED'*$
..... DELETE. (SEQPTR. (READL))
..... T'D ADVANN
..... MAKEDL. (PREM, DEF)
..... MAKEDL. (NEED, PREM)
..... EQU = ITSVAL. (EQUIV, SYSTEM)
..... W'R EQU .E. 0
..... EQU = LIST. (9)
..... NEWVAL. (EQUIV, EQU, SYSTEM)
..... E'L
..... NEWBOT. (DEF, EQU)
..... COND = 1
..... T'D LOOP1
R
RIF NOT CONCLUSION, SAVE PREMISE AND PREMISE PREDICATE
R
..... O'E
..... NEWBOT. (PREM, DEF)
..... TEMP = LIST. (9)
..... MAKEDL. (TEMP, PREM)
..... NEWBOT. (EQUIV .V. TRMNUM .LS. 18, TEMP)
..... COND = 3
..... T'D NEWPRM
..... E'L
..... E'L
R
RCHECK FOR BREAK BETWEEN TERMS
R
C'R WORD .E. $ .$.
W'R COND .NE. 2
PRNTP. (ERR3)
V'S ERR3 = $MISPLACED PERIOD$, 37777777777K
T'D ERRIN
..... O'E
..... NEWBOT. (TERM, PREM)
..... TRMNUM = TRMNUM + 1
..... COND = 4
..... T'D NEWTRM
..... E'L
R
RCHECK FOR BEGINNING OF NAME

```



```

-----
R
C'R WORD .E. $ -$
W'R COND .NE. 2
PRNTP. (ERR4)
V'S ERR4 = $MISPLACED HYPHENS, 37777777777K
T'D ERRIN
D'E
COND = 7
LENGTH = 0
NAME = $ $
NEWBOT. (TERM, PREM)
TRMNUM = TRMNUM + 1
T'D NEWORD
E'L

R
RCHECK FOR BEGINNING OF TERMINAL CHARACTER 'LITERAL'
R
C'R WORD .E. $ '$ .OR. WORD .E. $ *$ .OR. WORD .E.
1 $ /$
W'R COND .E. 3
P'T ERR10, WORD, LINE
V'S ERR10 = $H.MISPLACED '.,RCI,H.' IN LINE NUMBER.
1,13*$
T'D ZAPALL
D'E
BREAK = WORD
COND = 6
T'D NEWORD
E'L

R
RCHECK FOR COMMA AFTER PREDICATE
R
C'R WORD .E. $ , $
W'R COND .NE. 3
PRNTP. (ERR5)
V'S ERR5 = $MISPLACED COMMAS, 37777777777K
T'D ERRIN
D'E
COND = 4
T'D NEWORD
E'L

R
RCHECK FOR EQUALS, BEGINNING OF CONCLUSION
R
C'R WORD .E. $ = $
W'R COND .NE. 3 .AND. COND .NE. 1
PRNTP. (ERR6)
V'S ERR6 = $MISPLACED EQUALS SIGNS, 37777777777K
T'D ERRIN
D'E
EQUAL = 18
COND = 4
T'D NEWORD
E'L

R
RCHECK FOR TAB
R
C'R WORD .E. $ 1$
W'R COND .E. 3
PRNTP. (ERR13)
V'S ERR13 = $MISPLACED TABS, 37777777777K

```

```

-----
O'E T'O ERRIN
NEWBOT. (606060606072K, TERM)
COND = 2
T'O NEWORD
E'L
R
RCHECK FOR CARRIAGE RETURN
R
O'R WORD .E. $ 2$
W'R COND .E. 3
PRNTP. (ERR14)
V'S ERR14 = $MISPLACED CARRIAGE RETURNS$, 3777777777
177K
T'O ERRIN
O'E
NEWBOT. (606060606055K, TERM)
COND = 2
T'O NEWORD
E'L
R
RCHECK FOR $. INDICATES VARIABLE NEXT ENCOUNTERED
R SHOULD BE MARKED FOR NEED.
R
O'R WORD .E. 606060606053K
W'R COND .E. 3 .OR. EQUAL
PRNTP. (ERR25)
V'S ERR25 = $MISPLACED DOLLAR SIGNS$, 377777777777K
T'O ERRIN
O'E
DOLIND = 18
T'O NEWORD
E'L
R
RASSUME CHARACTER IS VARIABLE
R
O'E
W'R COND .E. 3
PRNTP. (ERR7)
V'S ERR7 = $MISPLACED VARIABLES$, 377777777777K
T'O ERRIN
O'E
COND = 2
VARIAB = ITSVAL. (WORD, VAR)
W'R EQUAL
W'R VARIAB .E. 0
PRNTP. (ERR8)
V'S ERR8 = $UNDEFINED VARIABLES$, 37777777
1777K
T'O ERRIN
E'L
NEWBOT. (VARIAB, TERM)
NEWBOT. (WORD, SVEVAR)
O'E
W'R VARIAB .E. 0
VARIAB = LIST. (9)
TEMP = LIST. (9)
MAKEDL. (TEMP, VARIAB)
NEWBOT. (WORD, VARIAB)
NEWVAL. (WORD, VARIAB, VAR)

```

```

O'R DOLIND
  POPTOP. (LSTNAM. (VARIAB))
O'E
  T'O ONLYON
E'L
  NEWBOT. (PREM, LSTNAM. (VARIAB))
  NEWBOT. (VARIAB, TERM)
  DOLIND = 0B
ONLYON
  E'L
    T'O NEWORD
  E'L
    R
    RIN CASE OF ERROR, CANON IS ERASED AND MAY BE RECONSTRUCTED
    R
    P'T ERR, LINE
    V'S ERR = $H' IN LINE NUMBER', I3*$
ERRIN
ZAPALL
  IRALST. (TERM)
  IRALST. (PREM)
  IRALST. (DEF)
  T'O LOOP1
  R
  RVARIOUS ERROR CHECKS FOLLOW
  R
  RCHECK TO SEE IF ALL NAMES ARE DEFINED
  R
EVAL
  ERRS = 0B
  DLIST = LSTNAM. (NAMES)
  SEQCHK = SEQRDR. (DLIST)
LOOP2
  NAME = SEQLR. (SEQCHK, F)
  DEFNUM = SEQLR. (SEQCHK, TEMP) .A. 777777K
  W'R F .G. 0
  W'R ERRS
    PRNTP. (COMM2)
    V'S COMM2 = $PLEASE DEFINE ABOVE PREDICATES.$, 7777
17777777K
  T'O LOOP1
  E'L
    PRINT COMMENT $ $
    PRNTP. (COMM1)
    V'S COMM1 = $CONSISTENT SET OF CANONS.$, 777777777777K
  T'O TYP0UT.
  E'L
    DEFCHK = ITSVAL. (DEFNUM, SYSTEM)
    W'R DEFCHK .E. 0
    ERRS = 1B
    P'T ERR9, NAME
    V'S ERR9 = $C6, H' UNDEFINED'*$
  E'L
    T'O LOOP2
  R
  RPRINT LIST OF PREDICATES.
  R
TYP0UT
  PRINT COMMENT $ $
  PRINT COMMENT $LIST OF DEFINED PREDICATES AND DEGREES.$
  PRINT COMMENT $ $
  I = 0
FNDNMB
  I = I + 1
  SEQCHK = SEQRDR. (LSTNAM. (NAMES))
  NAME = SEQLR. (SEQCHK, F)
  SPCNMB

```

```

DEFNUM = SEQLR. (SEQCHK, G)
PRMNUM = DEFNUM .RS. 18
DEFNUM = DEFNUM .A. 777777K
W'R F .G. 3, T'D PUTIN
W'R DEFNUM .E. 1
  P'T NOTE3, I, NAME, PRMNUM
  V'S NOTE3 = $I3,H'. -',C6,H'-',I4*$
  T'D FNDNMB

```

```

E'L
T'D SPCNMB

```

```

R
RINPUT OF SOURCE STRINGS AND 'NEED' FLAGS.
RA POINTER TO THE INPUT STRING IS USED IN
RTHE LIST, RATHER THAN THE INPUT ITSELF.
RTHE ADDRESS PORTION OF THE WORD CONTAINS THE
RNUMBER OF THE LAST CHARACTER INPUTTED
RAND THE DECREMENT CONTAINS THE NUMBER OF
RTHE FIRST. THOSE PARTS OF THE STRINGS
RDERIVED FROM THE CANONIC DEFINITIONS
RARE LEFT AS SINGLE CHARACTERS IN A SLIP
RCELL.

```

PUTIN

```

LIST. (MAXINP)

```

```

IRALST. (NAMES)

```

```

PRINT COMMENT $ $

```

```

PRINT COMMENT $INPUT OF SOURCE STRINGS.$

```

RETRY

```

PRINT COMMENT $TYPE FINAL PREDICATE.$

```

```

READIN. (NAME)

```

```

EQUIV = ITSVAL. (NAME, NAMES)

```

```

CHKNUM = EQUIV .RS. 18

```

```

EQUIV = EQUIV .A. 777777K

```

```

W'R EQUIV .E. 0

```

```

  P'T COMM4, NAME

```

```

  V'S COMM4 = $H'-',C6,H'- NOT FOUND*$

```

```

  T'D RETRY

```

```

E'L

```

```

LIST. (SEARCH)

```

```

IAP = 3B

```

```

LNECNT = 0

```

```

PRINT COMMENT $ $

```

```

PRINT COMMENT $TYPE -NONEED-, -NEED- OR -INPUT- FOR EACH TERM

```

1.\$

```

  THROUGH READY, FOR TRMNUM = 1, 1, TRMNUM .G. CHKNUM

```

```

  PRINT COMMENT $ $

```

```

  P'T COMM3, TRMNUM, NAME

```

```

  V'S COMM3 = $H'TERM NUMBER',I2,H' OF -',C6,H'-'*$

```

```

  READIN. (ANSWER)

```

```

  W'R ANSWER .E. $ NEED$

```

```

    NEWBOT. ($NEED$, SEARCH)

```

```

    T'D READY

```

```

  O'R ANSWER .E. $ NONEED$

```

```

    NEWBOT. ($PLEASE$, SEARCH)

```

```

    T'D READY

```

```

  O'R ANSWER .E. $ INPUT$

```

```

    INP = 1B

```

```

    PRINT COMMENT $ $

```

```

    PRINT COMMENT $INPUT STRING. EXTRA CARRIAGE RETURN

```

1INDICATES

```

  END.$

```

```

  PRINT COMMENT $ $

```

```

  SAVI = (LNECNT * 6 + 1) .LS. 18

```

```

  TEMP = LIST. (9)

```



```

R
RECURR  CHK1 = SEQRDR. (STACKB)
        CHK2 = SEQLR. (CHKR, F)
        CHK3 = SEQLR. (CHKR, TEMP)
        W'R F .G. 0, T'O OUTCHK
        W'R CHK1 .NE. DEFINE, T'O RECURR
        W'R LSTEQL. (SEARCH, CHK2) .E. 0
            W'R SWITCH .G. 0
            PRINT COMMENT $LEFT RECURSION DETECTED.$
        E'L
        T'O LUP0J2
    E'L
    T'O RECURR
R
RDEVELOPE 'NEED' LIST.
R
OUTCHK  FIND = LSSCPY. (SEARCH)
        TEMP1 = LIST. (9)
        MAKEDL. (TEMP1, FIND)
        NEWTOP. (SEQRDR. (FIND), TEMP1)
        TEMP = LSTNAM. (DEFINE)
        NEED = LSSCPY. (LSTNAM. (TEMP))
        PREM = SEQRDR. (TEMP)
        LCOK = SEQRDR. (FIND)
        LIST. (NONEED)
LUP0 3  PREMISE = SEQLR. (PREM, F)
        W'R F .G. 0, T'O PRTNED
        SEE = SEQLR. (LOOK, F)
        FNDTRM = SEQRDR. (PREMISE)
LUP0 5  VARIAB = SEQLR. (FNDTRM, G)
        W'R G .G. 0, T'O LUP0J3
        W'R G .E. 0
            VARIAB = TOP. (VARIAB)
            W'R SEE .E. $NEED$
            NEWBOT. (VARIAB, NEED)
        O'R F .E. 0
            NEWBOT. (VARIAB, NONEED)
    E'L
    T'O LUP0J5
PRTNED  W'R LEMPTY. (NONEED), T'O LUP0J6
        TEMP1 = POPTOP. (NONEED)
        FNDTRM = SEQRDR. (NEED)
LUP0 4  VARIAB = SEQLR. (FNDTRM, F)
        W'R F .G. 0, T'O PRTNED
        W'R VARIAB .NE. TEMP1, T'O LUP0J4
        DELETE. (SEQPTR. (FNDTRM))
        T'O LUP0J4
LUP0 6  IRALST. (NONEED)
        W'R SWITCH .LE. 1, T'O STRTSC
        TEMP1 = SEQRDR. (NEED)
LUP0 10 TEMP2 = SEQLR. (TEMP1, F)
        W'R F .G. 0, T'O STRTSC
        P'T NOTEJ, TEMP2
        V'S NOTEJ = $H'NEED ',RC1,H'.**$
        T'O LUP0J0
R
RGET CONCLUSION OF CANON.
R
STRTSC  NEWBOT. (FIND, STACK2)
        W'R SWITCH .G. 4, PRTLST. ($NEED$, NEED)

```

```

LUP007  CCNCL = SEQRDR. (TEMP)
        TERM = SEQLR. (CCNCL, F)
        W'R F .G. 0, T'O PRMCHK
        ECLIND = 1B
        IN = 0B
        INP = 0B
R
RGET NEXT TERM OF CONCLUSION.
R
        PIECE = SEQRDR. (TERM)
        T'O LUP011
LUP008  W'R IN, T'O GETINA
LUP009  IN = 0B
        W'R LEMPTY. (STACK1)
        W'R INP, T'O LUP007
        ECLIND = 0B
LUP011  CHAR = SEQLR. (PIECE, G)
        W'R G .G. 0
        W'R EDLIND
        INP = 1B
        O'E
        T'O LUP007
        E'L
R
RCHECK TO SEE IF SCAN HAS FAILED.
R
        W'R LEMPTY. (STACK2), T'O LUP001
        TEMP = STACK1
        STACK1 = STACK2
        STACK2 = TEMP
E'L
FIND = POPTOP. (STACK1)
SEE = LSTNAM. (FIND)
READS = POPTOP. (SEE)
W'R EDLIND
        SEQLR. (READS, F)
        W'R F .L. 0 .OR. INP
        NEWTOP. (READS, SEE)
        NEWBOT. (FIND, STACK2)
        INP = 1B
        T'O LUP009
        E'L
E'L
TEMP = CONT. (SEQPTR. (READS) + 1)
HCLDP = TOP. (TEMP)
HCLDT = BOT. (TEMP)
W'R G .L. 0
R
RTERMINAL CHARACTER IN CONCLUSION. CHECK STRING.
R
LUP015  W'R LEMPTY. (HOLDT), T'O NGOOD
        WORD = POPTOP. (HOLDT)
        STRND = 0B
        W'R WORD .L. 0
        W'R CHAR .E. WORD
        NEWBOT. (WORD, HOLDP)
        NEWBOT. (FIND, STACK2)
        NEWTOP. (READS, SEE)
        T'O LUP008

```

```

O'E
T'O NGOOD
E'L
O'E
OBJECT = CHARAC. (WORD)
W'R OBJECT .E. CHAR
LUP019  TEMP1 = WORD .A. 77777K6
W'R LEMPT. (HOLDP), T'O TRA0VR
TEMP = POPBOT. (HOLDP)
W'R TEMP .G. 0 .AND. (TEMP .A. 77777K)
1 .E. (WORD .RS. 18) - 1, T'O SKPOVR
NEWBOT. (TEMP, HOLOP)
TRA0VR  TEMP = TEMP1 .V. (WORD .RS. 18) - 1
SKPOVR  TEMP = TEMP + 1
NEWBOT. (TEMP, HOLDP)
W'R TEMP1 .GE. WORD .LS. 18, T'O JMPOVR
JMPOVR  WORD = WORD + 1K6
NEWTOP. (WORD, HOLDT)
W'R STRTND, T'O LUP015
NEWBOT. (FIND, STACK2)
NEWTOP. (READS, SEE)
T'O LUP008
O'R OBJECT .E. $00NULL$
STRTND = 1B
T'O LUP019
O'R OBJECT .E. $000END$
T'O LUP015
O'E
NGOOD  IRALST. (FIND)
W'R IN
CHAR = SAVECH
G = SAVEG
E'L
T'O LUP009
E'L
E'L
R
RVARIABLE IN CONCLUSION.
RCHECK TO SEE IF VARIABLE PREVIOUSLY DEFINED.
R
C'E
VARIAB = TOP. (CHAR)
DLIST = ITSVAL. (VARIAB, FIND)
W'R DLIST .NE. 0
SAVECH = CHAR
SAVEG = G
IN = 1B
G = -1
LUP021  DLIST = SEQRDR. (DLIST)
CHAR = SEQLR. (DLIST, F)
W'R F .G. 0, T'O GETOUT
W'R CHAR .L. 0, T'O LUP027
I = CHAR
LUP023  CHAR = CHARAC. (I)
I = I + 1K6
W'R CHAR .E. $000END$, T'O LUP021
W'R CHAR .E. $00NULL$, T'O LUP023
ALLRC = 1B
T'O LUP015
LUP027  ALLRC = 0B

```



```

T'D LUP015
GETINA      FIND = POPBOT. (STACK2)
            READS = POPTOP. (SEE)
            W'R ALLRC, T'D LUP023
            T'D LUP021
GETOUT      CHAR = SAVECH
            G = SAVEG
            NEWBOT. (FIND, STACK2)
            NEWTOP. (READS, SEE)
            T'D LUP009
            E'L
R
R VARIABLE IS NOT YET DEFINED, SO PROGRAM
R MUST SEARCH RECURSIVELY. SELECT PREMISE WITH WHICH
R TO SEARCH FOR VARIABLE.
R
            PRPNTR = TOP. (LSTNAM. (CHAR))
            PRNUM = TOP. (LSTNAM. (PRPNTR)) .A. 777777K
R
R CHECK OTHER TERMS (AND VARIABLES) IN CHOSEN PREMISE.
R
            LIST. (PUSHES)
            REMPTR = SEQRDR. (PRPNTR)
LUP031      TERM = SEQLR. (REMPTR, F)
            W'R F .G. 0, T'D PUSH1
            TEMP = TOP. (TERM)
            ZIEL = TOP. (TEMP)
R
R INSERT STRING FOR VARIABLE PRESENTLY SOUGHT.
R
            W'R ZIEL .E. VARIAB
            TEMP = LIST. (9)
            NEWBOT. (TEMP, PUSHES)
            TEMP2 = LIST. (9)
            NEWBOT. (TEMP2, TEMP)
            TEMP1 = LSSCPY. (HOLDT)
            NEWBOT. (TEMP1, TEMP)
            ABANDN. (TEMP1)
            O'E
R
R SEE IF OTHER VARIABLES PREVIOUSLY DEFINED.
R
            ISITDF = ITSVAL. (ZIEL, SEE)
            W'R ISITDF .NE. 0
            TEMP1 = LSSCPY. (ISITDF)
            TEMP2 = LIST. (9)
            TEMP = LIST. (9)
            NEWBOT. (TEMP2, TEMP)
            NEWBOT. (TEMP1, TEMP)
            ABANDN. (TEMP1)
            NEWBOT. (TEMP, PUSHES)
            O'E
R
R DECIDE WHETHER TO FLAG AS 'NEED' OR 'PLEASE'.
R
            W'R PRPNTR .NE. TOP. (LSTNAM. (TEMP)),
LUP035      1 T'D LUP037
            NDPTR = SEQRDR. (NEED)
            CKNEED = SEQLR. (NDPTR, F)
            W'R F .G. 0

```

```

LUP037           NEWBOT. ($PLEASE$, PUSHES)
                  O'R ZIEL .E. CKNEED
                  NEWBOT. ($NEED$, PUSHES)
                  O'E
                  T'O LUP035
                  E'L
                  E'L
                  E'L
                  T'O LUP031
E'L
R
RINFORMATION FOR RECURSION ASSEMBLED, SO SAVE STUFF
RFR THE PUSH.
R
PUSH1           SCMERC = 0B
                  W'R SWITCH .G. 1
                  P'T NOTE1, PRMNUM
                  V'S NOTE1 = $H'SCAN PUSH FOR',I3*$
E'L
T'O PUSHIT
PUSH2           SCMERC = 1B
                  W'R SWITCH .G. 1
                  P'T NOTEA, PRMNUM
                  V'S NOTEA = $H'PREMISE PUSH FOR',I3*$
E'L
PUSHIT           NEWTOP. (DEF, STACKA)
                  NEWTOP. (EQUIV, STACKA)
                  NEWTOP. (NEED, STACKA)
                  NEWTOP. (SEARCH, STACKB)
                  NEWTOP. (DEFINE, STACKB)
                  NEWTOP. (STACK1, STACKA)
                  NEWTOP. (STACK2, STACKA)
                  NEWTOP. (CONCL, STACKA)
                  NEWTOP. (PIECE, STACKA)
                  NEWTOP. (FIND, STACKA)
                  NEWTOP. (SEE, STACKA)
                  NEWTOP. (READS, STACKA)
                  NEWTOP. (HOLDP, STACKA)
                  NEWTOP. (HOLDT, STACKA)
                  NEWTOP. (CHAR, STACKA)
                  NEWTOP. (VARIAB, STACKA)
                  NEWTOP. (PRPNTR, STACKA)
                  NEWTOP. (ANSWER, STACKA)
                  NEWTOP. (SOMERC, STACKA)
                  NEWTOP. (EOLIND, STACKA)
                  EQUIV = PRMNUM
                  SEARCH = PUSHES
                  T'O LUP000
R
RCHECK WHETHER PREMISE CONDITIONS ARE SATISFIED.
R
PRMCHK           PIECE = SEQRDR. (DEFINE)
LUP051          W'R LEMPTV. (STACK1)
                  W'R LEMPTV. (STACK2), T'O LUP001
                  PRPNTR = SEQLR. (PIECE, F)
                  W'R F .G. 0, T'O ASSMBL
                  TEMP = STACK1
                  STACK1 = STACK2
                  STACK2 = TEMP
E'L

```

```

FIND = POPTOP. (STACK1)
SEE = LSTNAM. (FIND)
READS = POPTOP. (SEE)
PRMNUM = TOP. (LSTNAM. (PRPNTR)) .A. 77777K
DLIST = ITSVAL. (PRPNTR .A. 77777K, FIND)
R
RPREMISE HAS NOT BEEN PREVIOUSLY VERIFIED WHILE
RSEARCHING CONCLUSION.
R
W'R DLIST .E. 0
LIST. (PUSHES)
TERM = SEQRDR. (PRPNTR)
TOPS = SEQLR. (TERM, F)
W'R F .G. 0, T'O PUSH2
TOPS = TOP. (TOP. (TOPS))
DLIST = ITSVAL. (TOPS, FIND)
W'R DLIST .E. 0
R
RVARIABLE NOT YET DEFINED. INSERT 'NEED'
R
NEWBOT. ($NEED$, PUSHES)
W'R SWITCH .G. 0
P'T NOTED
V'S NOTED = $H-'NEED' REQUEST IN PREMISE CHEC
IK.--$
E'L
T'O LUP053
E'L
TEMP1 = LIST. (9)
NEWBOT. (TEMP1, PUSHES)
TEMP2 = LIST. (9)
NEWBOT. (TEMP2, TEMP1)
TEMP2 = LSSCPY. (DLIST)
NEWBOT. (TEMP2, TEMP1)
ABANDN. (TEMP2)
T'O LUP053
C'E
R
RPREMISE HAS BEEN PREVIOUSLY GENERATED IN SCAN
RCF CONCLUSION.
R
SOMERC = 0B
PUSHES = LSSCPY. (DLIST)
TERM = SEQRDR. (PUSHES)
TEMP = SEQRDR. (PRPNTR)
TOPS = SEQLR. (TERM, F)
TEMP3 = SEQLR. (TEMP, H)
W'R F .G. 0
W'R SOMERC, T'O PUSH2
IRALST. (PUSHES)
NEWTOP. (READS, SEE)
NEWBOT. (FIND, STACK2)
T'O LUP051
O'R F .L. 0
SOMERC = 1B
TEMP3 = TOP. (TOP. (TEMP3))
DLIST = ITSVAL. (TEMP3, FIND)
W'R DLIST .E. 0
P'T NOTED
SUBST. ($NEED$, SEQPTR. (TERM))

```



```

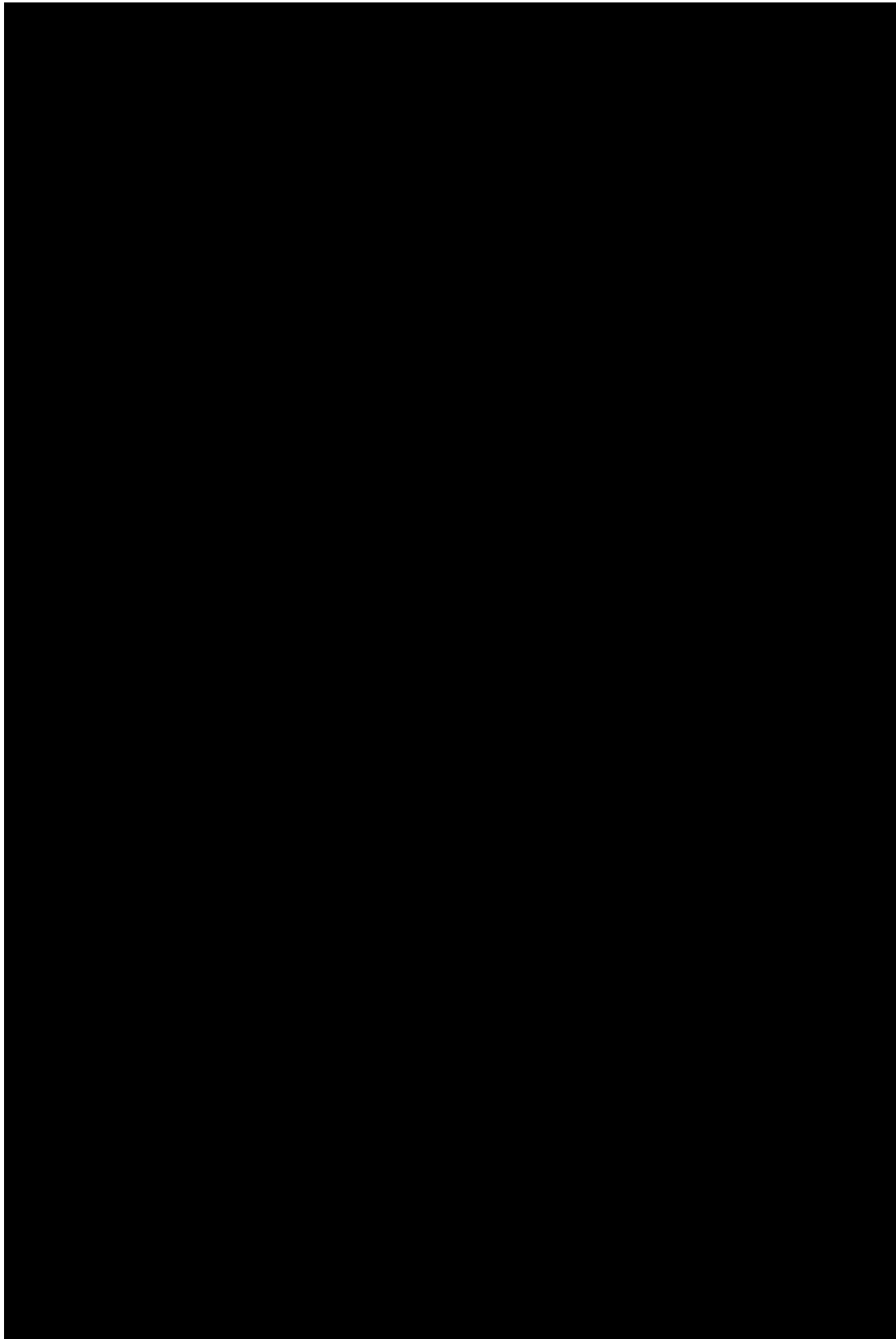
        E'L
        IRALST. (FIND)
        T'O ASSMBL
    E'L
    TEMP3 = LSSCPY. (ANTWRT)
    INLSTR. (TEMP3, (CONT. (TEMP .A.
1 77777K)) .RS. 18)
    IRALST. (TEMP3)
    O'R F .L. 0
    NEWBOT. (TEMP2, TEMP)
    E'L
    T'O LUP071
    E'L
    R
    RPOP-UP ROUTINE
    R
POP    W'R SWITCH .G. 2
        PRTLST. ($ANSWER$, ANSWER)
    E'L
    IRALST. (STACK1)
    IRALST. (STACK2)
    W'R LEMPTY. (STACKA), T'O THKGOD
    IRALST. (SEARCH)
    RTRN1 = ANSWER
    ECLIND = POPTOP. (STACKA) .E. 1
    SOMERC = POPTOP. (STACKA) .E. 1
    ANSWER = POPTOP. (STACKA)
    PRPNTR = POPTOP. (STACKA)
    VARIAB = POPTOP. (STACKA)
    CHAR = POPTOP. (STACKA)
    HCLDT = POPTOP. (STACKA)
    HCLDP = POPTOP. (STACKA)
    READS = POPTOP. (STACKA)
    SEE = POPTOP. (STACKA)
    FIND = POPTOP. (STACKA)
    PIECE = POPTOP. (STACKA)
    CENCL = POPTOP. (STACKA)
    STACK2 = POPTOP. (STACKA)
    STACK1 = POPTOP. (STACKA)
    DEFINE = POPTOP. (STACKB)
    SEARCH = POPTOP. (STACKB)
    NEED = POPTOP. (STACKA)
    EQUIV = POPTOP. (STACKA)
    DEF = POPTOP. (STACKA)
    G = 0
    W'R SWITCH .G. 1
        P'T NOTE2, EQUIV
        V'S NOTE2 = $H'POP BACK TO',I3,H'.!*$
    E'L
    W'R SOMERC, T'O POP2
    R
    RRETURN TO SCAN OF CONCLUSION AFTER PUSHING
    RFOR DEFINITION OF A VARIABLE.
    R
POP1   W'R LEMPTY. (RTRN1)
        IRALST. (RTRN1)
        IRALST. (FIND)
        INP = 0B
    W'R SWITCH .L. 4, T'O LUP009
    PRTLST. ($STACK1$, STACK1)

```

```

PRTLST. ($STACK2$, STACK2)
T'O LUP009
E'L
FNDCPY = LSSCPY. (FIND)
TEMP = SEQRDR. (FIND)
TEMP3 = SEQRDR. (FNDCPY)
LUP079 TEMP1 = SEQLR. (TEMP, F)
TEMP2 = SEQLR. (TEMP3, H)
W'R F .G. 0
T'O LUP080
D'E
W'R TEMP .E. READS
CPYHDP = TOP. (TEMP2)
CPYRDS = TEMP3
LINKS = CONT. (TEMP2 .A. 77777K) .RS. 18
E'L
T'O LUP079
E'L
R
RSAVE THE RETURN ANSWER, AND DEFINE VARIABLES
RAND PREDICATES AS GIVEN FROM PUSH.
R
LUP080 TEMP1 = POPTOP. (RTRN1)
TEMP3 = SEQRDR. (TEMP1)
TEMP2 = SEQRDR. (PRPNTR)
LUP081 TEMP4 = SEQLR. (TEMP3, H)
TEMP5 = SEQLR. (TEMP2, F)
W'R F .G. 0
NEWVAL. (PRPNTR .A. 77777K, TEMP1, FNDCPY)
ABANDN. (TEMP1)
NEWTOP. (CPYRDS, LSTNAM. (FNDCPY))
NEWBOT. (FNDCPY, STACK2)
T'O POP1
C'R H .L. 0
W'R TEMP4 .E. $NEED$
PRINT COMMENT '$NEED' ERROR.$
E'L
T'O LUP081
D'E
TMPVAR = TOP. (TOP. (TEMP5))
PRVDEF = ITSVAL. (TMPVAR, FNDCPY)
R
RVARIALE PREVIOUSLY DEFINED. COMPARE DEFINITIONS.
R
W'R PRVDEF .NE. 0
W'R LSTEQL. (PRVDEF, TOP. (TEMP4)) .NE. 0
IRALST. (FNDCPY)
IRALST. (TEMP1)
T'O POP1
E'L
D'E
R
RACD DEFINITION.
R
NEWVAL. (TMPVAR, TOP. (TEMP4), FNDCPY)
W'R VARIAB .E. TMPVAR
SUBST. (POPBOT. (TEMP4), LINKS)
CHK0 = LSSCPY. (TOP. (TEMP4))
NEWTOP. (LIST. (9), TEMP4)
W'R LEMPTY. (CHK0), T'O LUP081

```



```

THKGD2  W'R LEMPTY. (ANSWER)
        W'R ALLRC, T'O HERAUS
        W'R SOMERC
          PRINT COMMENT $SCAN COMPLETED. SYNTAX ERROR IN INP
1U1 STRING. $
        PRINT COMMENT $PART(S) OF INPUT OR NEED STRING(S) NOT SCANNED
1.$
          T'O LUP150
          O'E
            PRINT COMMENT $SCAN FAILED. SYNTAX ERROR IN INPUT
1STRING(S).$
          E'L
            PRINT COMMENT $NO TRANSLATED OUTPUT.$
LUP150  CHKNUM = 0
        MAX1 = 0
        CONCHK = SEQRDR. (SEARCH)
LUPERR  SEECHK = SEQLR. (CONCHK, F)
        W'R F .G. 0, T'O HERAUS
        CHKNUM = CHKNUM + 1
        W'R F .L. 0, T'O LUPERR
        I = SEQLR. (MAXCHK, F)
        OLDMAX = MAX1
        MAX1 = I .A. 777777K
        MAX2 = I .RS. 18
        PRINT COMMENT $ $
        W'R CHARAC. (I + 1K6) .E. $QJNULI $ .AND. MAX1 - MAX2
1 .L. 6
          P'T NOTE4, CHKNUM
          V'S NOTE4 = $H'INPUT TERM',I2,H' COMPLETELY SCANNED
1.'*$
          O'E
            MAX3 = CHARAC. (I)
            P'T NOTE5, CHKNUM, MAX3
            V'S NOTE5 = $H'LAST CHARACTER INSPECTED IN TERM'
1,I2,H' WAS ',RC1,H' IN MIDST OF FOLLOWING CONTEXT.'*$
            PRINT COMMENT $ $
            LINE1 = (MAX2 - 1)/6 - 2
            LINE2 = (OLDMAX + 5)/6 - 1
            LINE3 = (MAX1 - 1)/6
            THROUGH ERRLUP, FOR I = 0, 1, I .E. 5
            W'R LINE1 + I .LE. LINE2 .OR. LINE1 + I .G.
1 LINE3
          BUFFER(I) = 575757575757K
          O'E
            BUFFER(I) = INPUT(LINE1 + I)
          E'L
            CONTINUE
ERRLUP  P'T NOTE6, BUFFER(0),...,BUFFER(4)
          V'S NOTE6 = $5C6*$
          E'L
            T'O LUPERR
          C'E
            SOMERC = 18
            HOLD = POPTOP. (ANSWER)
            ENDCHK = SEQRDR. (HOLD)
            TEMP4 = SEQRDR. (SEARCH)
LUPSEE  SEECHK = SEQLR. (ENDCHK, F)
        TEMP5 = SEQLR. (TEMP4, H)
        W'R F .G. 0, T'O ALLOVR
        W'R TEMP5 .E. $PLEASE$, T'O LUPSEE

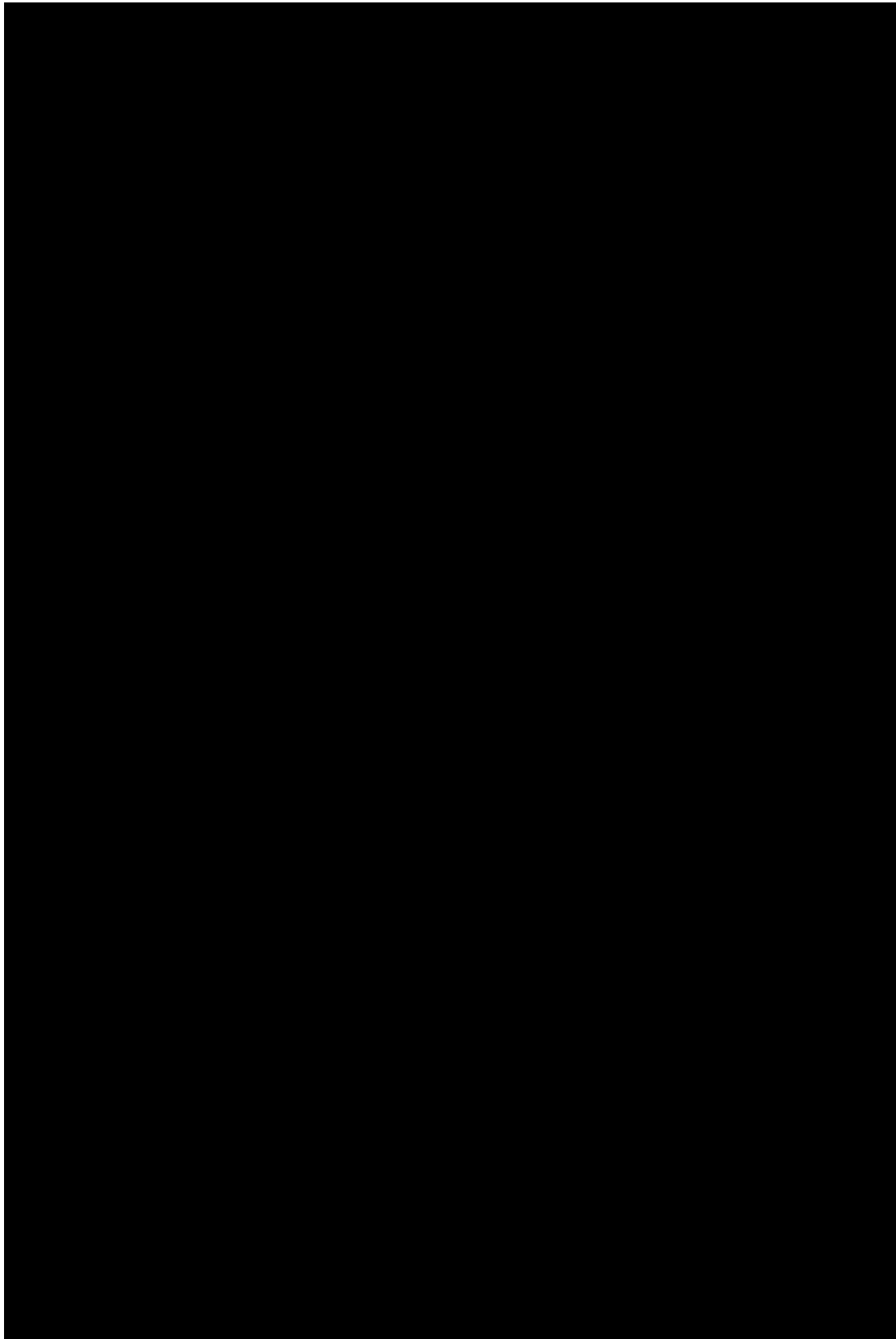
```



```

TEMP = BOT. (SEECHK)
W'R .NOT. LEMPTY. (TEMP)
TEMP1 = POPTOP. (TEMP)
W'R .NOT. LEMPTY. (TEMP), T'O THKGD1
TEMP2 = TEMP1 .RS. 18
TEMP3 = TEMP1 .A. 777777K
TEMP1 = CHARAC. (TEMP1)
W'R TEMP1 .NE. $OOOEND$ .AND. TEMP1 .NE.
I $OONULL$ .OR. TEMP3 - TEMP2 .G. 5, T'O THKGD1
E'L
T'O LUPSEE
E'L
R
RSCAN WAS SUCCESSFUL. PRINT OUT 'NEEDED' TERMS.
R
ALLOVR W'R ALLRC
PRINT COMMENT $ $
PRINT COMMENT $ADDITIONAL SUCCESSFUL SCAN.$
T'O ALLGNE
C'E
PRINT COMMENT $SCAN SUCCESSFUL.$
ALLRC = 18
ALLGNE PRINT COMMENT $TRANSLATED OUTPUT (IF ANY) FOLLOWS.$
CONCHK = SEQRDR. (SEARCH)
TRMNUM = 0
LUPOUT CONCL = SEQLR. (CONCHK, F)
SEECHK = SEQLR. (ENDCHK, G)
TRMNUM = TRMNUM + 1
W'R F .G. 0, T'O THKGD1
W'R CONCL .NE. $NEED$, T'O LUPOUT
PRINT COMMENT $ $
W'R SEECHK .E. $NEED$, PRINT COMMENT $'NEED' ERR$
P'T NOTE7, TRMNUM
V'S NOTE7 = $H'TERM NUMBER',12,H'.'*$
PRINT COMMENT $ $
SEECHK = TOP. (SEECHK)
INP = 08
LUP100 THROUGH LUP101, FOR I = 0, 1, I .E. 14
LUP101 BUFFER(I) = 575757575757K
BUFFER(14) = 777777777777K
I = 0
G = 30
W'RDCNT = 0
CHKEMP W'R G .LE. -6
G = 30
W'RDCNT = W'RDCNT + 1
E'L
W'R I .E. 80
CROUT PRNTP. (BUFFER (0))
T'O LUP100
E'L
W'R INP
LUP1.5 INP = 18
LUP1.7 TEMP1 = CHARAC. (TEMP)
TEMP = TEMP + 1K6
W'R TEMP1 .E. $OONULL$, T'O LUP107
W'R TEMP1 .E. $OOOEND$, T'O LUP109
T'O LUP113
D'E
LUP1.9 W'R LEMPTY. (SEECHK)

```



```
END READIN.  
CEND = 0  
DUMMY = $ $  
T'D ROLINE  
RDANS W'R WORD .R. $ -$.OR. WORD .E. $ $, T'D SKIPIT  
SKIPIT DUMMY = DUMMY .LS. 6 .V. WORD .A. 77K  
W'R EDLIND, FUNCTION RETURN  
T'D NEWORD  
E'N  
E'M
```

```

*****
M5364      5163      PRTLST      MAD FOR      M5364      5163      05,
EXTERNAL FUNCTION (NAME, LSTOUT)
N'S INTEGER
BCOLEAN EMPTY
E'D PRTLST.
PRINT COMMENT $ $
P'T NOTE6B, NAME, GETMEM. (0)
V'S NOTE6B = $C6,H' MEM=',16*$
I = 0
LIST. (STACK)
LISSNM = LSTOUT
START      NUMBER = ITSVAL. (LISSNM, STACK)
W'R NUMBER .NE. 0
P'T NOTE2, NUMBER
V'S NOTE2 = $H'LIST',13*$
AROUND    W'R LEMPTY. (STACK)
PRINT COMMENT $ $
IRALST. (STACK)
FUNCTION RETURN
D'E
S = POPTOP. (STACK)
POINT = POPTOP. (STACK)
NUMB = POINT .A. 777777K
POINT = POINT .RS. 18
W'R POINT .E. 1, T'D RETURN
T'D GOBACK
E'L
E'L
I = I + 1
NUMB = I
NEWVAL. (LISSNM, NUMB, STACK)
P'T NOTE3, NUMB
V'S NOTE3 = $H'BEGIN',13,H'.'*$
S = SEQRDR. (LISSNM)
L = LSTNAM. (LISSNM)
W'R L .NE. 0
PRINT COMMENT $DLIST.$
NEWTOP. (NUMB .V. 1K6, STACK)
NEWTOP. (S, STACK)
LISSNM = L
T'D START
RETURN    PRINT COMMENT $END DLIST.$
D'E
PRINT COMMENT $NO DLIST.$
E'L
GOBACK   W = SEQLR. (S, F)
W'R F .G. 0
P'T NOTE6, NUMB
V'S NOTE6 = $H'END',13,H'.'*$
T'D AROUND
C'R F .E. 0
W'R W .A. 7000007K5 .NE. 0, T'D READER
PRINT COMMENT $LIST NAME.$
NEWTOP. (NUMB, STACK)
NEWTOP. (S, STACK)
LISSNM = W
T'D START
C'E
READER   P'T NOTES, W, W
V'S NOTES = $H'.,C6,H' .',K12,H'.'*$

```

T'D GORACK
E'1
E'N

Bibliography

1. Cheatham, T. E. and Kirk Sattley, Syntax-Directed Compiling, Proceedings 1964 Spring Joint Computer Conference, pp. 31-57, American Federation of Information Processing Societies (1964).
2. Donovan, John J., Investigations in Simulation and Simulation Languages, Ph.D. Thesis, Yale University, New Haven, Connecticut; Fall, 1966.
3. Donovan, John J. and Henry Ledgard, A Formal System for the Specification of the Syntax and Translation of Computer Languages, M.I.T., 1967.
4. Post, E. L., "Formal Reductions of the General Combinatorial Decision Problem", American Journal of Mathematics, Vol. 65, pp. 197-215; 1943.
5. Shea, Dorothy, CTSS SNOBOL User's Manual, Project MAC Memo MAC-M-307-1, Project MAC, M.I.T., Cambridge, Mass.; October, 1966.
6. Smullyan, R. M., Theory of Formal Systems, Princeton University Press, Princeton, New Jersey; 1961.
7. Weizenbaum, J., "Symmetric List Processor", Communications of the ACM, Vol. 6, No. 9; September, 1963.
8. Weizenbaum, J., The Symbolic SLIP-Mad System, Project MAC, M.I.T., Cambridge, Mass; September, 1965.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-46

Each of the following should be identified by a checkmark:
Originating Department:

- Artificial Intelligence Laboratory (AI)
- Laboratory for Computer Science (LCS)

Document Type:

- Technical Report (TR) Technical Memo (TM)
- Other: _____

Document Information

Number of pages: 84 (91-images)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:

- Single-sided or
- Double-sided

Intended to be printed as :

- Single-sided or
- Double-sided

Print type:

- Typewriter Offset Press Laser Print
- InkJet Printer Unknown Other: _____

Check each if included with document:

- DOD Form Funding Agent Form Cover Page
- Spine Printers Notes Photo negatives
- Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description :	Page Number:
<u>IMAGE MAP: (1- 84) UN# 'EO TITLE + ACKN. PAGES, 3-37, UN# BLK, 79-84</u>	
<u>(85-91) SCANNING CONTROL, COVER, FUNDING AGENT, DOD, TRGT'S (3)</u>	

Scanning Agent Signoff:

Date Received: 12/11/95 Date Scanned: 12/22/95 Date Returned: 12/28/95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Massachusetts Institute of Technology Project MAC		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP None	
3. REPORT TITLE A Canonic Translator			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Bachelor's Thesis, Electrical Engineering, June 1967			
5. AUTHOR(S) (Last name, first name, initial) Alsop, Joseph W.			
6. REPORT DATE November 1967		7a. TOTAL NO. OF PAGES 84	7b. NO. OF REFS 8
8a. CONTRACT OR GRANT NO. Office of Naval Research, Nonr-4102(01)		9a. ORIGINATOR'S REPORT NUMBER(S) MAC-TR-46 (THESIS)	
b. PROJECT NO. NR 048-189		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
c. RR 003-09-01			
10. AVAILABILITY/LIMITATION NOTICES Distribution of this document is unlimited.			
11. SUPPLEMENTARY NOTES None		12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency 3D-200 Pentagon Washington, D. C. 20301	
13. ABSTRACT <p>This thesis presents an algorithm to recognize and translate sets of character strings specified by canonic systems. The ability of canonic systems to define the context sensitive features of strings and to specify their translation allows the algorithm to recognize and translate real computer languages. It is also applicable in other language systems.</p> <p>Canonic systems are discussed, and several examples of their use are given. The algorithm is described, and examples of canonic translation are presented using a program implementation.</p>			
14. KEY WORDS Canonic systems Machine-aided cognition Time-sharing Canonic translators Multiple-access computers Time-shared computers Computers On-line computers Translators			

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives**, using funds from the **Advanced Research Projects Agency** of the **United States Government** under Grant: **MDA972-92-J1029**.

The scanning agent for this project was the **Document Services** department of the **M.I.T. Libraries**. Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences**.

