To: Project MAC participants
From: J. J. Donovan, H. F. Ledgard
Subject: Canonic Systems and Their Application to Programming Languages

Abstract:

This paper presents two basic results: the use of established methods of recursive definition to
1. specify the syntax of computer languages (including context-sensitive requirements),
2. specify the translation of programs in one computer language into programs in another language.

The method can be used to write one specification for both the syntax of a language (e.g. a source language) and its translation into a target language (e.g. an assembler language). A syntactically legal program and its translation into the target language can then be generated using the specification. If the target language is understood, the semantics of the first language is specified.

The paper develops the method of recursive definition in conjunction with an example specifying the syntax of a limited subset of PL/I and its translation into IBM System/360 assembler language.

The application of the method to a generalized translator for computer languages is discussed.
1. **Introduction**

Our objective is to present a single method for expressing the syntax and translation of computer languages.

The objective to develop either methods for specifying the syntax or methods for specifying the translation of computer languages is not new. In response to the demand for numerous problem-oriented computer languages to meet the needs of diverse fields, there has been considerable activity to ease the effort required to implement a language.

Much of this activity has led to the development of methods for specifying, at least in part, the syntax of computer languages.\(^{(7-12)}\) The methods for specifying syntax have facilitated the description of computer languages to members of the computing field and have led to the development of syntax-directed translators.\(^{(22-25)}\) However, most of the methods for specifying syntax have been shown to be equivalent to context-free phrase structure grammars and hence inadequate\(^{(11,21,22)}\) for completely characterizing the syntax of computer languages. For example, some programming languages require that all statement labels in a program be different, that reference labels refer to existing statement labels, that the arithmetic type of a variable be declared, or that the dimensions of an array be declared before referring to an element within the array. These restrictions cannot be specified by a context-free grammar. We consider these restrictions to be syntactic in that programs violating these restrictions are never translated, but are rejected solely on their form. Debate as to whether these restrictions are syntactic or semantic is immaterial when we wish to specify both the syntax and translation of a language, because then all these restrictions must be satisfied.
Other activity has been directed to developing table-driven compilers and programming languages for expressing string transformations. The table-driven compilers have generally been limited to a particular type of target language and have required excessive detail in writing the specifications to fill the tables for a particular source language. The string transformation languages have been limited to special types of string transformations and have not been found generally useful for translating computer languages. Approaches to the formalization of semantics have also been made.

Here we present a single, formal method for specifying completely the syntax and translation of a computer language. The method is independent of both the source and target languages. The method uses an uncluttered, readable notation. The method recursively classifies sets of strings. The syntax of a computer language is characterized by specifying a set where each element is a syntactically legal program. The translation of a computer language is characterized by specifying a set of ordered pairs, where the first element of each pair is a legal program in the source language and the second element is a corresponding program in the target language that preserves the meaning of the source language program. If the target language is understood, the semantics of the source language has been specified.

The paper develops the method of recursive definition with an example specifying the syntax of a limited subset of PL/I and its translation into IBM System/360 assembler language. A discussion of the power of the method and of its application to a generalized translator is presented. An ordered set of appendices is also presented. The appendices present:

1) a brief summary of the notation for the method of recursive definition,
2) two programs in the subset of PL/I and their translation into System/360 assembler language, 3) a Backus-Naur Form specification of the syntax of the subset of PL/I, 4) a complete specification of the syntax of the subset
using the method of recursive definition, 5) a complete specification of the syntax of the subset and its translation into System/360 assembler language using the method of recursive definition, and 6) a derivation of a syntactically legal program in the subset and its translation into assembler language.

2. Basis of Formalization

The formalization for the method presented here evolved from Post's canonical systems, and hence will be called canonic systems. Smullyan used an applied variant of the canonical systems of Post in his definition of elementary formal systems. In class notes on the application of elementary formal systems to the definition of self-contained mathematical systems, Trenchard More modified the definition of elementary formal systems. Elementary formal systems (now called canonic systems in recognition of the earlier work by Post) were further modified to meet the definitional needs of computer languages and applied to the definition of syntax by Donovan. Canonic systems were later applied by Ledgard to specify the translation of computer languages. This paper is a synthesis of the last two works.

Canonic systems, which are equivalent to elementary formal systems, can be used to specify any recursively enumerable set. Smullyan used elementary formal systems as the basis for his entire study of formal mathematical systems and recursively enumerable sets. We use canonic systems to define two examples of recursively enumerable sets, the set of syntactically legal programs comprising a computer language, and the set of ordered pairs specifying the translation of programs in one language to programs in another language. We may feel confident that canonic systems can specify any programming language, or more generally, any algorithm that a machine can perform. This confidence is a direct consequence of Church's thesis.
which asserts that the logical notion of "recursive function" or "recursively enumerable set" (which is encompassed by canonic systems) is capable of fully describing the intuitive concept of an algorithm.

3. **Canonic Systems**

A canonic system is a finite sequence of rules for recursively defining sets. The elements of the sets are strings of symbols selected from some finite alphabet. Each rule is called a canon. A canon generally has the form

\[ a_1 \text{ set } A_1 \vdash a_2 \text{ set } A_2 \vdash \ldots \vdash a_n \text{ set } A_n \vdash b \text{ set } B \]

which may be interpreted informally:

If "a_1" is a member of the set named "set A_1", and "a_2" is a member of the set named "set A_2", ..., and "a_n" is a member of the set named "set A_n", then we can assert that "b" is a member of the set named "set B".

The "a_i" and "b" represent symbols from the finite alphabet; the "set A_i" and "set B" are the names of the sets defined.

In the remainder of this section we will elaborate on this notation.

A synopsis of the notation is given in Appendix 1. and may be used as a reference throughout the text. The notation will be developed by a series of examples taken from the canonic system specifying the syntax of a subset of PL/I, called Little PL/I. This subset includes limited forms of PL/I GO TO statements, IF statements, label assignment statements, label declaration statements, and arithmetic assignment statements. The Backus-Naur Form description of the subset is given in Appendix 3. Two example syntactically legal programs in the subset are given in Appendix 2. One of these examples is repeated here:
Q: PROCEDURE;
    DECLARE LX LABEL;
    L: I = I+1A*IB-IC;
    LX=L;
    GO TO CHECK;
    M: I = I+1;
    LX=M;
    CHECK: IF I<LIMIT THEN GO TO LX;
    END Q;

We define the syntax of a language as the set of rules for specifying
the strings that can be recognized by a translator and translated into
some other language. The set of rules excludes strings that the translator
would reject solely on their form. The syntax of Little PL/I has the
following restrictions, which for all practical purposes make Little PL/I
context-sensitive and therefore impossible to completely characterize in
Backus-Naur Form:

1. Different declarations of the same identifier are in error, i.e.
   a. The lists of fix-pt variables, statement labels and declared
      label variables for a program must be mutually disjoint, and
   b. the label before PROCEDURE must not occur within the procedure
      block.
2. The label after END must be identical to the label before PROCEDURE.
3. All statement labels must be different.
4. The identifier in a GO TO statement must refer to an existing
   statement label or a declared label variable.
5. The identifier on the left hand side of the "=" in a label assignment
   statement must refer to a declared label variable; the identifier
   on the right hand side of the "=" must refer to an existing statement
   label or a declared label variable.

We begin our discussion of the notation for canonic systems by specifying
a set named "letter". The canons for this set are as follows:

| A letter | (1) |
| B letter | (2) |
| ... | ... |
| Z letter | (26) |
These canons may be read:

(From no premises) we can assert that the symbol "A" is a member of the set named "letter".
(From no premises) we can assert that the symbol "B" is a member of the set named "letter".
And so on.

The canons specify a set named "letter" comprising the capital letters of the English alphabet. The sign "\( \vdash \)" is the assertion sign. The strings "A letter", "B letter", ..., and "Z letter" are conclusions. The capital English letters A through Z are members of the object language. The underlined, lower case characters are predicates. A predicate, here "letter", is the name of a set.

The set named "identifier" may be specified in terms of the set named "letter":

\[
\begin{align*}
\langle 1 \rangle \text{ letter} & \vdash \langle 1 \rangle \text{ identifier} \\
\langle 1 \rangle \text{ letter} & \vdash \langle 2 \rangle \text{ letter} & \vdash & \langle 1 \rangle \langle 2 \rangle \text{ identifier} \\
\vdots & \\
\langle 1 \rangle \text{ letter} & \vdash \langle 2 \rangle \text{ letter} & \vdash & \cdots & \vdash \langle 8 \rangle \text{ letter} & \vdash \langle 1 \rangle \langle 2 \rangle \langle 3 \rangle \langle 4 \rangle \langle 5 \rangle \langle 6 \rangle \langle 7 \rangle \langle 8 \rangle \text{ identifier} 
\end{align*}
\]

(27) (28) (34)

These canons may be read:

If "\( \langle 1 \rangle \)" is a member of the set named "letter", then we can assert that "\( \langle 1 \rangle \)" is a member of the set named "identifier".

If "\( \langle 1 \rangle \)" is a member of the set named "letter" and "\( \langle 2 \rangle \)" is a member of the set named "letter", then we can assert that "\( \langle 1 \rangle \langle 2 \rangle \)" is a member of the set named "identifier".

And so on.

The canons* specify a set consisting of identifiers of one to eight capital

---

* Actually, these canons should properly be called canon schema, which denote instances of canons.
English letters. The lower case (possibly subscripted or superscripted) English letters $\lambda_1$ through $\lambda_8$ are variables that represent members of the set named "letter". The strings "$\lambda_1$", "$\lambda_1\lambda_2$" are terms. A term is a string of variables or symbols from the object alphabet (e.g. "$\lambda_1\lambda_2\lambda_3\lambda_4\lambda_5\lambda_6\lambda_7\lambda_8$"), The sign " $\vdash$ " is the conjunction sign. The strings "$\lambda_1$ letter", "$\lambda_2$ letter", "$\lambda_3$ letter" are premises. The " $\vdash$ " before and " $\vdash$ " separates premises, all of which must be satisfied to assert the conclusion. The strings "$\lambda_1$ letter", "$\lambda_1$ identifier", "$\lambda_2$ letter", "$\lambda_1\lambda_2$ identifier" are remarks.

The vocabulary used to describe a canon is summarized as follows:

\[
\begin{array}{cccc}
\lambda_1 & \text{letter} & \vdash & \lambda_1 & \text{identifier} \\
\text{variable} & \text{variable} & \text{term} & \text{predicate} & \text{term} & \text{predicate} \\
\text{premise} & \text{conclusion} & \text{remark} & \text{remark} \\
\hline
\text{canon}
\end{array}
\]

To demonstrate the property of recursion in canons, we define a set named "list". The set named "list" is needed in the canonic system for the syntax of Little PL/I to specify the requirement that the list of all reference labels for a program must be contained in the list of all statement labels. The canons for the set named "list" are as follows:

\[
\begin{align*}
\lambda_1 & \vdash \lambda_1 \text{list} \\
\lambda_1 \text{identifier} & \vdash \lambda_1, \lambda_1 \text{list} \\
\lambda_1 \text{list} & \vdash \lambda_1 \text{list} \\
\hline
\lambda_2 \text{list} & \vdash \lambda_2 \text{list} \\
\lambda_1 \lambda_2 \text{list} & \vdash \lambda_1 \lambda_2 \text{list} \\
\hline
\end{align*}
\]

We can use canons (1) through (37) to show that the string "A,BB,A," is
a member of the set named "list". Using canon (1) we can assert that "A" is a member of the set named "letter". Using canon (27) we can assert "A" is a member of the set named "identifier". Using canon (36) we can assert that "A," is a member of the set named "list". Similarly, using canons (2), (28), and (36) we can assert that "BB," is a member of the set named "list". Since the premises "A, list" and "BB, list" have been asserted, we can use the instance of canon (37)

\[ A, \text{list} \vdash BB, \text{list} \vdash A, BB, \text{list} \]

to assert that "A, BB," is a member of the set named list. Using canon (37) again (recursively) and letting "a" denote the list "A, BB," and "b" denote the list "A," we can assert that "A, BB, A," is a member of the set named "list".

The repeated use of the same predicate names in the above canons strongly suggest the use of two abbreviations:

1. If \( t_1, t_2, \ldots, t_n \) are terms denoting members of the same set named "S", the remarks
   \[ t_1 \downarrow S \vdash t_2 \downarrow S \vdash \ldots \vdash t_n \downarrow S \]
   may be abbreviated
   \[ t_1 \uparrow \ldots \uparrow t_n \downarrow S \]

2. If \( C_1, C_2, \ldots, C_n \) are conclusions with identical premises \( Q \), the canons
   \[ Q \vdash C_1 \]
   \[ Q \vdash C_2 \]
   \[ \vdots \]
   \[ Q \vdash C_n \]
   may be abbreviated
   \[ Q \vdash C_1 \uparrow C_2 \uparrow \ldots \uparrow C_n \]

Thus the canons for "letter" may be abbreviated with abbreviation 2:

\[ \vdash A \text{ letter} \vdash B \text{ letter} \vdash \ldots \vdash Z \text{ letter} \]
or further abbreviated with abbreviation 1:

\[ \vdash A \vdash B \vdash \ldots \vdash Z \text{ letter} \]

The canons for "identifier" may be abbreviated:

\[ A \vdash B \vdash \ldots \vdash \text{letter} \downarrow \vdash \text{identifier} \]

This canon may be read:

If "\( \text{l}_1 \)"", "\( \text{l}_2 \)"", \ldots , and "\( \text{l}_8 \)" are members of the set named "\text{letter}",
then we can assert that "\( \text{l}_1 \)”, "\( \text{l}_2 \)”, \ldots , and "\( \text{l}_8 \)” are members of the set named "\text{identifier}".

We also introduce the following abbreviation:

3. In cases where many canons have some common premises, a block structure abbreviation may be used. The common premises are stated once and understood to be added to the premises of all canons within the scope of the common block.

This abbreviation greatly eased the writing of the specification for the syntax of the simulation language GPSS. (4) There was no need to incorporate this abbreviation into the canonic system of Little PL/I. An abbreviation of two or more canons is formally called an edict. However, we will usually refer to edicts as canons.

Thus far we have specified only sets of 1-tuples, i.e., predicates of degree 1. For a complete specification of Little PL/I, we will have to specify sets of n-tuples for n greater than 1, i.e., predicates of degree n, \( n > 1 \). We use the notation

\[ \langle a_1 \langle a_2 \langle \ldots \langle a_n \rangle \rangle \ldots \rangle \text{ where } a_1, a_2, \ldots, a_n \text{ are terms}, \]

to denote an ordered n-tuple. The sign "\( \langle \)" separates elements of an ordered

* To obtain the given abbreviation for "identifier", we must first change the premises of each of the eight unabbreviated canons to "\( \text{l}_1 \text{ letter} \downarrow \)
\( \text{l}_2 \text{ letter} \downarrow \ldots \downarrow \text{l}_8 \text{ letter} \)". This change does not affect the validity of the canons.
n-tuple. For the requirement that the list of reference labels for a Little PL/I program be contained in the list of statement labels, we specify a set named "in". The predicate "in" names the set of all ordered pairs such that the first element is a list of identifiers and the second element is a list of identifiers contained in the first list. The set named "in" is defined:

\[(38) \quad a \uparrow b \uparrow c \text{ list } \vdash <b,abc> \text{ in} \]
\[(39) \quad <a\ell> \text{ in } \vdash <b\ell> \text{ in } \vdash <ab\ell> \text{ in} \]

To suggest more strongly the relationship among the elements of an n-tuple, we allow premises and conclusions of the form

\[<a_1 a_2 \ldots a_n> \text{ text1 text2 \ldots textn-1} \]

to be written

\[a_1 \text{ text1} a_2 \text{ text2} \ldots \text{ textn-1} a_n \]

Thus the canons for "in" become

\[(38) \quad a \uparrow b \uparrow c \text{ list } \vdash b \text{ in } abc \]
\[(39) \quad a \text{ in } \ell \vdash b \text{ in } \ell \vdash ab \text{ in } \ell \]

These canons may informally be read:

If "a", "b", and "c" are lists, then we can assert that the list "b" is contained in the list "abc".

If the list "a" is contained in the list "\ell" and the list "b" is contained in the list "\ell", then we can assert that the list "ab" is contained in the list "\ell".

The following conclusions can be asserted from canons (1) through (39):

\[A, \text{ in } A, B, \quad A, A, \text{ in } A, \quad R, S, \text{ in } S, T, R, \quad \text{ALPHA, in } \text{ALPHA, BETA}, \]
Finally, we extend the use of the symbol "\( \uparrow \)" to abbreviate the writing of n-tuples by allowing remarks of the form

\[
\text{a}_1 \text{ text}_1 \text{ a}_2 \text{ text}_2 \ldots \text{ text}_n \text{ a}_n \uparrow \text{ a}'_1 \text{ text}_1 \text{ a}'_2 \text{ text}_2 \ldots \text{ text}_n \text{ a}'_n
\]

to be abbreviated

\[
\text{a}_1 \uparrow \text{ a}'_1 \text{ text}_1 \text{ a}_2 \uparrow \text{ a}'_2 \text{ text}_2 \ldots \text{ text}_n \uparrow \text{ a}'_n
\]

Thus canon (39)

\[
a \uparrow b \uparrow ab \uparrow
\]

may be abbreviated

\[
a \uparrow \text{ b } \uparrow \text{ ab } \uparrow
\]

(39)

4. Canonic System Specification of Syntax

This section is concerned with the motivation and development of a canonic system for the syntax of Little PL/I, including context-sensitive requirements. Our approach will be to specify eventually a set of n-tuples named "PL/I program". Each member of this set will be a syntactically legal Little PL/I program. The entire canonic system specification of this set is given in Appendix 4. The numbers of the BNF productions of Appendix 3. and the canons of Appendix 4. correspond in that productions and canons with corresponding numbers specify corresponding syntactic constructions.

The sets of letters and digits are specified by canons 1. and 2. of Appendix 4. An identifier in Little PL/I is a string of one to eight letters. Canon 3. specifies the set of identifiers. Similarly, the set of unsigned integers, whose members are strings of one to ten digits, is specified by canon 4. Little PL/I has only fixed-point arithmetic variables. Canons 5.1 and 5.2 specify the set of fixed-point variables. Canons 6. and 7.1 specify the set of labels and label variables.

To specify the restriction that a reference label in a GO TO statement
be contained in the list of statement labels or that all label variables
for a program be contained in the list of declared label variables, we define
the sets named "list" and "in" (canons 7.2 through 7.6). The canons for
"list" specify a set of lists, where each list is a sequence of identifiers
separated by commas. The canons for "in" specify a set of ordered pairs of
lists, where each identifier in the first list is contained in the second list.

To specify the restrictions that the sets of fixed-point variables,
statement labels, declared label variables, and the procedure label for a
program must be mutually disjoint, we define the set of ordered pairs named
"disjoint" (canons 7.10 through 7.13). The first element of each ordered
pair is a list of identifiers; the second element of each pair is a list
of identifiers, none of which appears in the first list. For example, the
ordered pair 
\[<A,B,C,<D,E,F,>>\]
\[<D,E,F,>]\] is a member of this set. To define the
set named "disjoint", we first define a set named "differ" (canons 7.7 through 7.9).
Members of the set named "differ" are ordered pairs, in which the first
element is an identifier and the second element is a different identifier.

Canons 8.1 through 18.4 specify the constructions for Little PL/I
primaries, GO TO statements, relational operators, boolean expressions,
IF statements, label assignment statements, arithmetic assignment statements, and
label DECLARE statements. For example, with every GO TO statement we
keep track of its reference label (to check later if it is in the list of
statement labels) or keep track of its label variable (to check if it is in
the list of declared label variables). Thus we define the canon for a
GO TO statement:

\[\text{label} \to \text{GO TO } \ell; \text{goto stm with ref label } \ell, \text{label var}\] (9.1)

This canon specifies a set of 3-tuples named "goto stm with ref label-label var".
The first element of a 3-tuple is a GO TO statement, the second element
the reference label for the GO TO statement, the third element the label
variable for the GO TO statement. The canon has the following instance:

A label \( \vdash \) GO TO A; goto stm with ref label A, label var \( \land \)

or

A label \( \vdash \) <GO TO A; A \( \land \) \( \!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
\!
then we can assert that "ss'" is a stm seq with stm labels "s's'", ref labels "r_s', label vars "v', declared label vars "v_v', and fix-pt vars "v'".

The premise "s disjoint l_s'" insures that the statement labels for each statement sequence are different.

Canon 20.2

\[
\begin{align*}
&s \text{ stm seq with stm labels } s \text{ ref labels } r \text{ label vars } \ell_v \text{ decl label vars } \ell_v \text{ fix-pt vars } v \\
&\quad \Downarrow \quad \ell_v \text{ disjoint } v \Downarrow \quad v \text{ disjoint } s \Downarrow \quad (20.2) \\
&\text{ and } s \text{ legal stm seq with stm labels } s \text{ vars } v_v \\
\end{align*}
\]

specifies a syntactically legal statement sequence. The premises "\(\ell_v \text{ disjoint } v_s'\)" and "\(v \text{ disjoint } l_s'\)" insure that the lists of declared label variables, fixed-point variables, and statement labels for a Little PL/I program are mutually disjoint. The premises "\(r \text{ in } s\)" and "\(v \text{ in } \ell_v\)" insure that all reference labels refer to existing statement labels and that the label variables used in the label assignment and GO TO statements are declared.

Finally, canon 21.

\[
\begin{align*}
&\ell \text{ label } \Downarrow \quad s \text{ legal stm seq with stm labels } s \text{ vars } v \Downarrow \quad l \text{ disjoint } s \Downarrow \quad (21.) \\
&\quad \Downarrow \quad \ell : \text{PROCEDURE } s \text{ END } l \text{; PL/I program} \\
\end{align*}
\]

specifies a syntactically legal PL/I program. The premise "\(\ell \text{ disjoint } s\)'" insures that the procedure label "\(\ell\)" is not used within the statement sequence for the program. The set named "PL/I program" is our desired set, the set of 1-tuples such that each member of the set is a syntactically legal Little PL/I program.

We thus state our first basic result:

1. The language of canonic systems can be used to specify (exactly) the syntax of a computer language.
5. **Canonic System Specification of Translation**

We know that theoretically, at least, the translation of programs from one computer language to programs in another computer language is a process that *can* be expressed by a canonic system. We can view every translator as specifying a function from one set of strings to another. Since the translation of programs is a task performed on computers, we can assert that the function specifying the translation is recursively enumerable and hence can be specified by a canonic system. The ordered pairs defining the function comprise the recursively enumerable set. The first element of each ordered pair is a program in one language, the second element its translation into the target language.

We use this last fact to motivate our development. We will develop the specification for the translation of a language by

a. modifying the specification of the syntax of the language to distinguish further the semantically different strings (e.g., the set of arithmetic operators +, -, *, and / will be split into two sets, one for the addition operators + and - and one for the multiplication operators * and /), and

b. appending to each n-tuple specifying a translatable string one more element specifying a corresponding string in some other language that preserves the meaning of the first string.

Instead of eventually specifying a set of 1-tuples comprising the set of syntactically legal programs (as we did in the previous section), we will eventually specify a set of ordered pairs. The first element in each pair will be a syntactically legal program, the second element its translation into the target language.

As in the previous section, we will illustrate our approach by example. The syntactic specification of Little PL/I will be modified to specify not only the syntax of Little PL/I but also its translation into IBM System/360 assembler language. A complete specification of the syntax and translation of Little PL/I is given in Appendix 5. Two example syntactically legal programs and their translations specified by Appendix 5.
are given in Appendix 2. As in the example translations, the canons of Appendix 5 specify comments entries with the assembler statements so that (hopefully) the reader will not have to be familiar with IBM System/360 assembler language to understand the translation.

In the succeeding paragraphs of this section we present a detailed discussion of the canons of Appendix 5. The reader may wish to omit this discussion. The techniques used in forming the canons of Appendix 5 may be grasped by comparing the correspondingly numbered canons of Appendices 4 and 5. For example, canons 9.1 and 9.2 of Appendix 4 give the canons specifying the syntax of a GO TO statement, and canons 9.1' and 9.2' of Appendix 5 give the canons specifying the syntax of a GO TO statement and its translation into assembler language. Following this section we continue with a discussion of the application of canonic systems to a generalized translator.

Consider first canon 9.1 of Appendix 4:

\[
\label \quad \text{GO TO } \ell; \quad \text{goto stmt with ref label } \ell, \text{ label var } A
\]  

(9.1)

The assembler language statement for a GO TO statement such as "GO TO A;", where A is a reference label, is simply

\[
B \quad A \quad \ast\text{BRANCH TO } A
\]

where "B" is the operation code for an assembler branch statement, "A" is the symbolic address of the first assembler statement for the referenced Little PL/I statement*, and "\ast\text{BRANCH TO } A" is a comments entry. We may specify this translation by modifying canon 9.1:

\[
\ell \quad \text{label } \quad \text{GO TO } \ell; \quad \text{goto stmt with ref label } \ell, \text{ label var } A \quad \text{assembler stmts}
\]

(9.1)

B \quad \ell \quad \ast\text{BRANCH TO } \ell

* We assume that identifiers occurring in a Little PL/I statement will not be changed upon translation into assembler statements so that the same identifiers may be used.
The modified predicate "goto stm with ref label-label var-assembler stms" names a set of 4-tuples, where the elements of a 4-tuple are

1. a GO TO statement
2. the reference label for the GO TO statement
3. the label variable for the GO TO statement
4. the assembler statements for the GO TO statement.

The instance of canon 9.1' for the GO TO statement "GO TO A;" is simply

\[
\text{A label } \triangleright \text{ GO TO A; goto stm with ref label A, label var A assembler stms}
\]

\[
\text{B A } \ast \text{BRANCH TO A}
\]

Similarly, canon 9.2 of Appendix 4, the canon for a GO TO statement with a label variable, is modified to specify its translation:

\[
\text{I label variable } \triangleright \text{ GO TO I; goto stm with ref label A, label var A assembler stms}
\]

\[
\text{L I,A } \ast \text{LOAD ADDRESS STORED IN A} \quad (9.2')
\]

\[
\text{BR 1 } \ast \text{BRANCH TO THIS ADDRESS}
\]

We next consider canon 11 of Appendix 4:

\[
p \triangleright p' \text{ primary with fix-pt var v} \downarrow v' \uparrow r \text{ rel op}
\]

\[
\text{prp' boolean exp with fix-pt vars v} \downarrow v' \quad (11)
\]

The assembler language statements for a boolean expression such as "I<F5" in an IF statement

\[
\text{IF } \langle \text{boolean exp} \rangle \text{ THEN } \langle \text{non-declare statement} \rangle
\]

might be as follows:

\[
\text{L 1,I } \ast \text{LOAD I}
\]

\[
\text{C 1,=F'5' } \ast \text{COMPARE WITH } \text{=F'5'}
\]

\[
\text{BNL Z } \ast \text{BRANCH IF NOT LOW TO Z}
\]

\[
\text{. (translation for non-DECLARE statement following the boolean expression)}
\]

\[
\text{Z EQU } \ast \quad \ast \text{SYMBOLIC ADDRESS OF Z}
\]

For the first three assembler statements we must know:

a. the operand entries (I and =F'5') for the primaries (I and 5) of the boolean expression
b. the branch operation code (BNL) for the relational operator (<)
c. the symbolic address (Z) of the assembler statement to which control should be passed if the boolean expression is not true.

These requirements necessitate that we

a. modify the canon for primaries to specify the operand entry for each primary
b. modify the canon for relational operators to specify the operation code for each relational operator
c. add a premise to canon 11. to specify a new label
d. modify the conclusion of canon 11. to carry the new label for later use (canon 12') in specifying the EQU assembler statement to follow the assembler statements for the non-DECLARE statement.

Thus canons 8.1, 8.2, 10., and 11. are modified to give:

\[
\begin{align*}
i & \text{unsigned integer} \quad i \text{ primary with fix-pt var} \wedge \text{operand code } \neq F'Re' \quad (8.1') \\
v & \text{fix-pt var} \quad v \text{ primary with fix-pt var} \wedge \text{v. operand code } v \quad (8.2') \\
\langle \langle \phi = \psi \rangle \rangle & \text{relop with branch code BNL\textup{\textasciitilde}BNL\textup{\textasciitilde}BNL message NOT LOW\textup{\textasciitilde}NOT \text{ LOW\textasciitilde} NOT \text{ EQUAL\textasciitilde} NOT \text{ HIGH} \quad (10') \\
p \mathbin{\downarrow} p' & \text{primary with fix-pt var} \wedge p' \text{ primary with fix-pt var} \wedge \text{v. operand code } x \mathbin{\downarrow} x' \\
\mathbin{\downarrow} r & \text{relop with branch code b} \wedge \text{message } g \mathbin{\wedge} \ell \text{ label} \\
& \mathbin{\downarrow} \text{prp' boolean exp with fix-pt vars } v v' \wedge \text{branch label } \ell \text{ assembler stms } \\
\text{L} & \mathbin{\downarrow} 1, x \quad \ast \text{LOAD } x \\
\text{C} & \mathbin{\downarrow} 1, x' \quad \ast \text{COMPARE WITH } x' \\
\text{b} & \mathbin{\downarrow} \ell \quad \ast \text{BRANCH IF } g \text{ TO } \ell
\end{align*}
\]

The modifications to canon 12. to specify the translation of an IF statement readily follow:

\[
\begin{align*}
b & \text{boolean exp with fix-pt vars } v \wedge \text{branch label } \ell \text{ assembler stms } s_a \mathbin{\downarrow} (12') \\
& \mathbin{\downarrow} \text{s non-DECLARE stmt with ref label } \ell_r \wedge \text{label vars } \ell_v \wedge \text{fix-pt vars } v' \\
& \mathbin{\downarrow} \text{assembler labels } a \wedge \text{assembler stms } s'_a \mathbin{\downarrow} \ell_a, \text{ disjoint } \ell_a \\
& \mathbin{\downarrow} \text{IF } b \text{ THEN } s \text{ if stmt with ref label } \ell_r \wedge \text{label vars } \ell_v \wedge \text{fix-pt vars } v v' \\
& \mathbin{\downarrow} \text{assembler labels } a \ell'_a, \text{ assembler stms } \\
& \mathbin{\downarrow} s'_a \wedge s_a \\
\ell & \mathbin{\downarrow} \text{EQU } * \quad \ast \text{SYMBOLIC ADDRESS OF } \ell_a
\end{align*}
\]

Here the premise "\text{\ell}_a, \text{disjoint } \ell'_a" insures that the new branch label differs from any that occur in the translation of the non-DECLARE statement "s".
The element "assembler labels" in the predicate for an IF statement is specified so that the assembler labels may later be specified to be different from the other assembler labels and PL/I identifiers for a program.

Canons 13.1 and 13.2 are easily rewritten to specify the translation of a label assignment statement:

\[
\begin{align*}
\text{v label var } & \vdash \ell \text{ label} \\
\text{v} & = \ell; \text{ label assign stmt with ref label } \ell, \text{ label vars v, assembler stmt } \\
\text{MVC } v & = A(\ell) \quad \text{*SET v TO ADDRESS OF } \ell \\
\text{v label var } & \vdash v = v'; \text{ label assign stmt with ref label label vars v, assembler stmt } \\
\text{MVC } v & = v'; \quad \text{*SET v EQUAL TO v'} \\
\end{align*}
\]

We next consider canons 14., 15., and 16., the canons for an arithmetic expression:

\[
\begin{align*}
\text{r } + & \text{ f } \quad \text{arith op} \\
\text{p primary with fix-pt vars v } & \vdash \text{ p arith exp with fix-pt vars v} \\
\text{a } \cdot & \text{ a' arith exp with fix-pt vars v } \vdash v' \quad \text{o arith op} \\
\text{a } \cdot & \text{ a' arith exp with fix-pt vars v} \\
\end{align*}
\]

We cannot immediately add to each of these canons the elements specifying the correct assembler language statements. In the evaluation of arithmetic expressions, the operations of multiplication and division are carried out before the operations of addition and subtraction. We may specify this requirement for a left to right evaluation of arithmetic expressions by:

a. separating the arithmetic operators (+, -, *, and/) into two classes, one for the addition operator (+ and -) and one for the multiplication operators (* and /)—canons 14.1' and 14.2'
b. defining a new construct "term" that consists of a sequence of two or more primaries separated by multiplication operators (* and /)
c. re-defining an arithmetic expression as a sequence of one or more primaries or terms separated by addition operators (+ and -)
d. specifying that the left-most primary or term in an arithmetic expression be evaluated first, in machine register 1—canons 15.1' through 15.4' for result in register 1
e. specifying that succeeding primaries in an arithmetic expression be directly added or subtracted to machine register 1—canon 15.5'  
f. specifying that succeeding terms in an arithmetic expression be formed first in machine register 3 (canons 15.2' and 15.3' for result in register 3) before being added or subtracted to machine register 1—canon 15.6'.

For example, canons 15.1' and 15.5' specify "I+1" as a legal arithmetic expression with assembler statements:

\[
\begin{align*}
L & \quad 1,1 \\
A & \quad 1,=P'1' \\
& \quad *LOAD 1 \\
& \quad *ADD =P'1'
\end{align*}
\]

Canons 15.1', 15.2', 15.3' and 15.6' specify "A+B*C*D" as a legal arithmetic expression with assembler statements:

\[
\begin{align*}
L & \quad 1,A \\
& \quad *LOAD A \\
L & \quad 3,B \\
& \quad *LOAD B \\
M & \quad 2,C \\
& \quad *MULTIPLY BY C \\
M & \quad 3-1,D \\
& \quad *MULTIPLY BY D \\
AR & \quad 1,3 \\
& \quad *ADD REGISTERS
\end{align*}
\]

The remaining canons of Appendix 4 are readily modified. Canon 16., the canon for an arithmetic assignment statement, is modified to specify the assembler store instruction appended to the assembler statements for an arithmetic expression. Canon 17., the canon for a DECLARE statement, requires no modification. Canons 18.1 through 19.4, the canons for a non-DECLARE statement and a statement sequence of one statement, need only be rewritten to carry the associated assembler labels and assembler statements. (Canon 20.2 must also be modified to append the assembler EQU statement to the assembler statements for a labeled non-DECLARE statement.)

Canons 20.1 and 20.2, the canons for a statement sequence of two or more statements, are also modified to carry the associated assembler labels and assembler statements. The premise "\( \not_a \text{ disjoint}_a \)" added to canon 20.1' insures that the assembler labels differ from each other and the premise "\( \not_a \text{ disjoint}_a \not_v \)" added to canon 20.2' insures that the assembler labels differ from the other identifiers in a Little PL/I program.
Canons 21.1' and 21.2'

\[
\begin{align*}
&\vdash \text{set of vars with assembler data stms} \\
&\quad s_d \quad \vdash \text{set of vars with assembler data stms} \quad v \quad \vdash \text{v identifier} \quad \vdash \text{v, disjoint v} \quad \text{(21.1')} \\
&\vdash \text{v, set of vars with assembler data stms} \\
&\quad s_d \quad \vdash \text{v, set of vars with assembler data stms} \quad v \quad \vdash \text{v identifier} \quad \vdash \text{v, disjoint v} \quad \text{(21.2')} \\
&\vdash \text{v, set of vars with assembler data stms} \\
&\quad v \quad \vdash \text{DS \quad F \quad *STORAGE FOR v_1} \\
\end{align*}
\]

are new. These two canons specify the assembler data storage statements for a set of variables. For example, if "I" were the only variable in a PL/I program, the following instances of canons 21.1' and 21.2' would specify the storage statements for "I":

\[
\begin{align*}
&\vdash \text{set of vars with assembler data stms} \\
&\quad s_d \quad \vdash \text{set of vars with assembler data stms} \quad \vdash \text{I identifier} \quad \vdash \text{I, disjoint I} \quad \text{v in v'} \\
&\vdash \text{I, set of vars with assembler data stms} \\
&\quad \text{I \quad DS \quad F \quad *STORAGE FOR I} \\
\end{align*}
\]

Finally, canon 21., the canon for a syntactically legal Little PL/I program, is modified to give

\[
21.3' \vdash \text{label} \quad \vdash \text{s legal stmt seq seq with stmt labels} \quad \vdash \text{l vars v assembler stms s_a} \\
\quad l, \quad \text{disjoint l vars v in v'} \quad \vdash \text{l : PROCEDURE; s END l; PL/I program with translation} \\
* \\
* \text{ASSEMBLER LANGUAGE PROGRAM FOR l} \\
* \\
\begin{align*}
&\text{BALR \quad 15,0} \quad \text{SET REGISTER 15 AS BASE REGISTER} \\
&\text{USING \quad *,15} \quad \text{INFORM ASSEMBLER OF USE OF R15 AS BASE REGISTER} \\
&\text{SVC \quad 0} \quad \text{RETURN TO SUPERVISOR} \\
&\text{END \quad l} \quad \text{TERMINATE ASSEMBLY} \\
\end{align*}
\]

The added premise "v' set of vars with assembler data stms s_d" specifies the assembler data storage statements for the variables of the legal statement sequence.
The set named "PL/I program with translation" is our desired set. This predicate names the set of all ordered pairs such that the first element of each ordered pair is a syntactically legal Little PL/I program and the second element is its translation into assembler language.

We thus state our second basic result:

2. The language of canonic systems can be used to specify the translation of programs in one computer language into programs of another computer language.

If the predicates needed to specify the syntax of a source language are properly modified, the language of canonic systems can be used to write one specification for both the syntax of a source language and its translation into a target language.

6. Derivation of a Syntactically Legal Program and its Translation into a Target Language

We have defined the rules for constructing the canonic system of Little PL/I in English. However, this canonic system could have been defined by a second canonic system; the elements of the second canonic system would formalize the rules for constructing canons, premises, terms, etc. In fact, starting with the canonic system of Little PL/I, there is an unending series of canonic systems, each of which specifies the canonic system previously written.

Canons of the second canonic system are called statutes. Statutes are read using the same rules as were used to read canons. The sign "\|" is the assertion sign of the second canonic system. We have informally given the definitions of a derivation, a list of premises, a conclusion, and a canon of the first canonic system. The predicates "derivation", "list of premises", "conclusion", and "canon" name sets in the second
canonic system that formalize these definitions. Two statutes of the second canonic system are:

\[ \begin{align*}
\text{derivation} & \quad \text{list of premises} \quad \text{conclusion} \\
\text{derivation} & \quad \text{list of premises} \quad \text{conclusion} \quad \text{amid} \\
\text{derivation} & \quad \text{list of premises} \quad \text{conclusion} \\
\end{align*} \]

The predicate "canon" names the set of all canons of the canonic system of the syntax of Little PL/I. The predicate "amid", like the predicate "in", specifies that one list (here a list of premises) be contained in another list (here a list of conclusions previously derived). These statutes formalize the notion of a "derivation" and are read:

1. From no premises, we can assert that the null string can be derived.
2. If a) a derivation "d" has been derived and b) the list of premises "l" in the canon "l \models l_1" is amid the derivation "d", then the string "l_1" can be appended to the derivation.

For example, by successively using the following instances of the set named "canon" in statute (2):

\[
\begin{align*}
\text{A letter} & \\
\text{A letter} \vdash \text{A identifier} & \\
\text{A identifier} \vdash \text{A label} & \\
\text{A label} \vdash \text{GO TO A; goto stm with ref label A, label var A} & \\
\end{align*}
\]

we obtain the following member of the set named "derivation"

\[
\text{A letter} \quad \text{A identifier} \quad \text{A label} \quad \text{GO TO A; goto stm with ref label A, label var A}
\]

This is a derivation of the string "GO TO A". (This string could alternately have been derived by applying the two rules of inference, substitution and Modus Ponens, to the canons of the canonic system of Little PL/I.)

An example of a member of the set named "derivation", using the canonic system of Appendix 4, as members of the set named "canon", is generated in Appendix 6. This derivation is a formal proof that the string
P: PROCEDURE;
A: I = I+1;
   IF I<5 THEN GO TO A;
END P;

is a PL/I program. Statute (2) above is used recursively to generate the
right-hand column of \( \ell_1 \) through \( \ell_{50} \). The right-hand column of lines
\( \ell_1 \) through \( \ell_{50} \) is the member of the set named "derivation". The third
column gives the number of the canon of Appendix 4. used to add the conclusion
given in the right-hand column to the derivation. The second column gives
the numbers of conclusions previously derived that are used as premises to
the canon of Appendix 4. The first column gives a number to the newly
derived conclusion. For example, line \( \ell_1 \) is asserted using the instance
of Statute 2:

\[
\begin{align*}
\text{derivation} & \quad \text{list of premises} \quad \text{first letter for fix-pt var conclusion} \\
& \quad \text{first letter for fix-pt var canon} \\
& \quad \text{first letter for fix-pt var derivation}
\end{align*}
\]

Since all premises of this statute are true, we can assert the conclusion
"I first letter for fix-pt var derivation". Lines \( \ell_1 \) and \( \ell_2 \) are asserted
by using an instance of statute 2 where "d" denotes "I first letter for fix-pt var c", "A" denotes "A", and "l_1" denotes "l_1 letter". Similarly, lines \( \ell_1 \) through
\( \ell_{50} \) are generated.

Appendix 7. presents a member of the set \textit{derivation} of the canonic
system of the canonic system of the translation of Little PL/I using the
canons for the translation (Appendix 5.) as members of the set "canon".
The member of the set named "derivation" given in Appendix 7. is a formal
proof of the PL/I program "A" and its translation into System/360 assembler
language.

The derivations of Appendices 6. and 7. lead to two observations.
First, the derivation provides us with a structural description of a derived
string. By a \textit{structural description} of a string we mean the sequence of
rules (here a sequence of canons) used in forming the string; the rules used give us certain information about the construction of the string. For example, using a derivation of a string, we may construct its "syntactic tree". Consider the portion of the derivation from Appendix 6:

If we consider only the first elements of each derived n-tuple, this derivation provides a structural description for the string "IF I<5 THEN GO TO A;" that may be represented in form of a syntactic tree:

The tree can be constructed by scanning the derivation from left to right and constructing the corresponding tree from the bottom up. In general, if we designate only certain elements of a canonic system, the portions of the derivation of a string for these elements provides us with a structural description of the string with respect to these elements.
Second, no additional mechanism was needed to derive the translation of the syntactically legal program. The string representing the translation into assembler language was derived like any other string needed in the derivation.

7. Application to a Generalized Translator

An important characteristic of a canonic system is that it is "generative". That is, like a Backus-Naur Form specification, a canonic system defines a string by generating the string using some rules of inference.

To use a canonic system specification in a generalized translator (i.e., translator that is independent of both source and target languages), an algorithm to recognize strings specified by a canonic system must be devised. No algorithm for recognizing strings specified by a canonic system is presented in this paper. An explicit recognition algorithm has been developed (6) for canonic systems and is being implemented. It is hoped that with this implementation canonic systems may be used in the recognition and translation of computer languages for a generalized translator.

A possible implementation of such a generalized translator is indicated in Figure 1. Here we depict two procedure segments and one data segment. The procedure segment named Encoding Program accepts a canonic system and forms it into a data segment to be referenced by the translator procedure segment. The translator procedure uses the encoding of the canonic system as a data base, recognizes source language programs specified by the canonic system, and outputs the associated translation into the target language.
The present recognition algorithm (6) was initially devised to recognize strings of a source language using the canonic system of the syntax of a source language as a data base. Using the same recognition algorithm and a canonic system for the syntax of a source language and its translation into a target language as a data base, no additional mechanism is needed to construct the translation of a recognized program. In general, it appears that no additional mechanism is needed to construct the translation for any algorithm that has the ability to recognize one element of an ordered n-tuple and construct another element.

Further, as indicated by Appendices 4. and 5., the language of canonic systems appears to provide an uncluttered notation for expressing the syntax and translation of a language. The writer of a canonic system need not think in terms of character scanning algorithms, parsing algorithms for syntactic trees, symbol tables, literal tables, and many other details that plague translator writing.
The generalized translator could be written to first recognize a string and then output its derivation. This derivation could be used to construct the structural description (e.g., syntactic tree) specified by designated elements of the derivation.

8. Discussion

Our approach in developing the canonic system for the syntax and translation of Little PL/I was to modify the canonic system of the syntax alone by adding to each n-tuple defining a translatable string another element specifying its translation. We found that we further had to modify the canonic system of the syntax in two ways. First, we had to rewrite the canons for certain syntactic constructs (e.g., the canon for arithmetic operators) so that semantically different parts were separated (e.g., the canon for arithmetic operators had to be separated into two canons, one for addition operators and one for multiplication operators). Second, we had to define a few additional elements and canons (e.g., the canons for assembler data storage statements). We also found it convenient to define elements such as message fields.

In practice, a canonic system different from Appendix 4 was initially devised to specify the syntax of Little PL/I. This canonic system was often found inconvenient to rework to specify the translation. The reworking led to the canonic systems of Appendices 4, and 5. We note that we did not attempt to devise independent canonic systems for the syntax and for the translation of Little PL/I. In a specification of a translation, a specification of a syntax is implicit and cannot be separated.

Many features of PL/I were omitted in choosing the subset Little PL/I. We have ignored the specification of allowable spacing of Little PL/I programs. Spacing, as well as card format has been specified for another computer language (4) using canonic systems. Some of the other omitted features
(e.g., nested arithmetic expressions) would have required only minor modifications to the specifications of Appendices 4. and 5. The use of identifiers containing up to thirty-six characters would have required that the identifiers in a program be changed upon translation to assembler language, where identifiers of only eight or less characters are allowed. The use of both fixed and floating point variables was initially considered, but was omitted because many additional canons and several additional elements were needed to specify the translation of arithmetic expressions. The inclusion of these features would not have added any new ideas to our development. The large number of other PL/I features, such as the assortment of data types, block structure, procedure references and definitions, and input/output were disregarded. It is certainly within the capacity of canonic systems to specify both the syntax and translation of these features.

However, an equally important issue is whether canonic systems provides a natural and concise specification of these features. A canonic system for the complete syntax of only one language, the simulation language GPSS \(^{(4)}\) has been written. Judging from the modest size (14 type-written pages) of this specification and the ease with which the partial syntactic specification of Appendix 4. was written, we feel that a canonic system for the complete syntax of PL/I would not be unduly large.

On the other hand, it is clear that a complete specification of the syntax of PL/I and its translation to assembler language would be large. We feel that this largeness is due to three principal factors. First, the PL/I language is complicated. Second, on some features PL/I and IBM System/360 assembler language are poorly matched. For example, many modifications to the canons of Appendix 5. would have been required to specify the assembler statements for the evaluation of arithmetic expressions containing both fixed and floating point variables. For any source and target languages, if
a canonic system of the syntax of the source language and its translation to the target language were written before finally defining the languages, it is likely that we could better match the languages. The unwieldiness of portions of the canonic system would clearly indicate where the languages were ill-matched. Third, we feel that a large part of the unwieldiness is due to the shortcomings of assembler language as a general-purpose target language. We feel strongly that if a better set of language primitives were devised, the specification of the translation process, using the new language as a target language, would be greatly eased.

It is important to develop languages whose descriptions are concise. The Backus-Naur Form specification of Appendix 3. and the associated five English sentences of Section 4. describing the context-sensitive requirements provide a very concise description of the syntax of Little PL/I. In their present form canonic systems will not replace Backus-Naur Form and the English language for describing programming languages to people. However, canonic systems, although yielding larger specifications, provide complete descriptions in a precise language that a machine can be instructed to understand with present techniques (6). Moreover, the language of canonic systems, like Backus-Naur Form, is readable.

We wish to point out two additional features of the canonic systems of Appendices 4. and 5. First, barring any inadvertent errors, the canonic systems describe a set of PL/I programs and assembler language programs that will run on a computer when translated by a PL/I compiler or System/360 assembler. Second, the specification of the comments entries in the assembler language statements was provided not only to aid the reader. The comments are meaningful context-sensitive strings in the English language. The specification of these strings was handled as easily as the specification of the strings in assembler language. The specification
of the strings in the English language illustrates the use of canonic systems to specify the entire operation of a translator, including the specification of meaningful comments. Moreover, it suggests the capacity of canonic systems to handle communication and translation in languages other than computer programming languages. We have not explored this enticing area.

Canonic systems are applicable to the definition of an "abstract" syntax (13) of a language. An abstract syntax of a language is a description of the syntax of a language that is independent of any actual representation of the symbolic expressions in the language. By 1) omitting the canons specifying only symbols in the object language (e.g., the canons for "letter" and "digit"), and 2) using the application of the abstract syntactic functions in place of elements specifying strings in object language (e.g., writing the canon for a Little PL/I GO TO statement as "/label GO (label with ref label \ label var \ label var \ label var /" , where "GO" is a function to be specified in defining a "concrete" syntax of Little PL/I), a canonic system could be developed to provide an exact specification of an "abstract" syntax of a language.

As mentioned earlier, the results of this paper apply to any recursively enumerable set. Any function or relation that is recursively enumerable can be specified by a canonic system. Canonic systems can be used to express language and string transformations of a much more different nature than given here. The facility with which comments entries were specified suggests many uses, for example, in handling inter-terminal computer communication. We do not know to what extent canonic systems can practically be used to express more varied algorithms than those for string transformations.
We have used canonic systems to present a single method for specifying the syntax and translation of computer languages. To ease further the specification of the translation of computer languages it might be desirable to use a target language other than another computer language. This new target language would have a set of primitives in which the constructions in a large class of languages can readily be expressed. This target language has not been developed.
REFERENCES

The work presented in this paper has evolved from the following works:

1. Post, Emil L. Formal Reductions of the General Combinatorial Decision Problem, Am. J. Math, Vol. 65, pp. 197-217, 1943. This work presents a formal system named canonical systems and demonstrates that every system in canonical form can formally be reduced to a system in normal form.

2. Smullyan, Raymond M. Theory of Formal Systems, Princeton University Press, Princeton, New Jersey, 1961. This work describes a variant (elementary formal systems) of the canonical systems of Post as the basic formalization for a study of formal mathematical systems and recursively enumerable sets.

3. More, Trenchard, Class notes for course EAS 313b Applied Discrete Mathematics, Yale University, New Haven, Conn., spring 1965. In this work the definition of elementary formal systems were modified to parallel More's definition of propositional complexes (ref. 30) and applied to the study of various mathematical systems.

4. Donovan, John J. Investigations in Simulation and Simulation Languages, Ph.D. dissertation, Yale University, New Haven, Conn., Fall 1966. Elementary formal systems were further modified (now called canonic systems in recognition of earlier work of Post) to meet the definitional needs of programming languages and applied to the definition of the syntax of computer languages.

5. Ledgard, Henry F. The Transformation of Source Language Programs into Applicative Expression Form, Term paper for course 6.688, Linguistic Structures, M.I.T., January 1967. Here canonic systems were applied to specify the translation of computer languages.

The following reference describes an algorithm that uses a canonic system specification of a language as a data base to recognize strings specified by a canonic system and generate their translation.


The following references describe other formalizations used to characterize computer languages:

The following references describe formalizations used to characterize the semantics of computer languages:


The following papers describe programming languages for expressing string transformations:


The following paper describes a computer program for expressing transformations on natural language input strings:

The following references (and ref. 11) point out inadequacies of past specifications in specifying the syntax of programming languages:


The following references (and ref. 16) describe generalized translators that have been implemented as table-driven compilers:


The following references describe to an extent the syntax of PL/I:


Definitions of the logical terminology used in this paper may be found in the following works:


The following manuals were used in developing the example subset of Little PL/I and defining its translation to IBM System/360 assembler language:


38. -----, A Programmer's Introduction to the IBM System/360 Architecture, Instructions, and Assembler Language, Student text, Form C20-1646-1.
To provide the reader with a concise reference, the canonic systems notation used in this paper will be presented by a series of short examples.

### Basic Constructions:

<table>
<thead>
<tr>
<th>Canon(s)</th>
<th>Interpretation</th>
<th>New Construction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( \neg A ) letter</td>
<td>(From no premises, we can conclude that) ( A ) is a member of the set named &quot;letter&quot;.</td>
<td>&quot;( \neg )&quot; is the assertion sign; &quot;letter&quot; is a conclusion; &quot;letter&quot; is the name of a set and ( A ) is an element in the set. The name of the set, here &quot;letter&quot;, is called a predicate.</td>
</tr>
<tr>
<td>2. ( \neg B ) letter</td>
<td>&quot;( B )&quot; is a member of the set named &quot;letter&quot;.</td>
<td></td>
</tr>
<tr>
<td>3. ( \neg C ) letter</td>
<td>&quot;( C )&quot; is a member of the set named &quot;letter&quot;.</td>
<td></td>
</tr>
<tr>
<td>4. ( \neg ) letter</td>
<td>If ( \neg \psi ) is a member of the set named &quot;letter&quot;, then we can conclude that ( \psi ) is a member of the set named &quot;letter&quot;.</td>
<td></td>
</tr>
<tr>
<td>5. ( \neg ) identifier</td>
<td>If ( \neg \psi ) is a member of the set named &quot;identifier&quot; and ( \psi ) is a member of the set named &quot;identifier&quot;, then &quot;( \psi )&quot; is a member of the set named &quot;identifier&quot;.</td>
<td></td>
</tr>
<tr>
<td>6. ( \neg ) differ</td>
<td>The ordered pair ( \langle A, B \rangle ) is a member of the set named &quot;differ&quot;. The set named &quot;differ&quot; consists of ordered pairs whose elements are different.</td>
<td></td>
</tr>
<tr>
<td>7. ( \neg ) differ</td>
<td>The ordered pair ( \langle A, C \rangle ) is a member of the set named &quot;differ&quot;.</td>
<td></td>
</tr>
</tbody>
</table>

### Interpretation

(From no premises, we can conclude that) "A" is a member of the set named "letter" and "B" is a member of the set named "letter" (same as canons 1., and 2.). The small letter \( \psi \) is a variable denoting any member of the set named "letter".

The name of the set, here "letter", is called a predicate.

If \( \psi \) is a member of the set named "identifier", then conclusion "\( \psi \)" is a member of the set named "identifier".

If \( \psi \) is a member of the set named "identifier" and \( \lambda \) is a member of the set named "identifier", then conclusion "\( \lambda \)" is a member of the set named "identifier".

The ordered pair \( \langle A, B \rangle \) is a member of the set named "differ" (same as canons 6. and 7.).

### New Construction(s)

\( \neg \) letter + \( \neg \) letter

\( \neg \) differ + \( \neg \) differ

### Abbreviations and Alternate Notations:

<table>
<thead>
<tr>
<th>Canon(s)</th>
<th>Interpretation</th>
<th>New Construction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8. ( \neg A ) letter + ( \neg B ) letter</td>
<td>&quot;A&quot; is a member of the set named &quot;letter&quot; and ( B ) is a member of the set named &quot;letter&quot; (same as canons 1., and 2.).</td>
<td>&quot;( \neg )&quot; after the &quot;( \neg )&quot; separates conclusions, all of which can be asserted if the premises (here none) are satisfied.</td>
</tr>
<tr>
<td>9. ( \neg A + \neg B ) letter</td>
<td>&quot;( A )&quot; and ( B ) are members of the set named &quot;letter&quot; (same as canon 8.).</td>
<td>&quot;( + )&quot; is used to separate two or more elements that are members of the same set.</td>
</tr>
</tbody>
</table>
| 10. \( \neg A \) differ B | The ordered pair \( \langle A, B \rangle \) is a member of the set named "differ" (same as canon 6.). | Alternate notation for "\( \langle A, B \rangle \) differ". In general, the notation \( <a_1, a_2, \ldots, a_n> \) is an alternate form for \( <a_1, a_2, \ldots, a_n> \) text...

where \( <a_1, a_2, \ldots, a_n> \) is an n-tuple and "text1 text2 ... textn\" is name of the set in which the n-tuple is a member.

11. \( \neg A \) differ B | The ordered pairs \( \langle A, B \rangle \) and \( \langle A, C \rangle \) are members of the set named "differ" (same as canons 6. and 7.). | "\( + \)" is used to separate the elements of two or more ordered pairs that are members of the same set. This notation is extended to handle n-tuples.
Appendix 2  Two Syntactically Legal Programs in Little PL/I and their Translation in to IBM SYSTEM/360 Assembler Language

Legal Program P

P: PROCEDURE;
   A: I = I+1;
   IF I < 5 THEN GO TO A;
END P;

Translation for P

* ASSEMBLER LANGUAGE PROGRAM FOR P
*
* BALR 15,0      *SET REGISTER 15 AS BASE REGISTER
P USING #,15      *INFORM ASSEMBLER OF USE OF R15 AS BASE REGISTER
A EQU *            *SYMBOLIC ADDRESS FOR PL/I LABEL A
   L 1,1            *LOAD I
   A 1,'P'1'        *ADD =F'1'
   ST 1,1            *STORE RESULT IN I
   L 1,1            *LOAD I
   C 1,'F'5'        *COMPARE WITH =F'5'
   BNL Z            *BRANCH IF NOT LOW TO Z
   B A              *BRANCH TO A
   Z EQU *            *SYMBOLIC ADDRESS OF Z
   SVC 0             *RETURN TO SUPERVISOR
   *
   I DS F            *STORAGE FOR I
   END P             *TERMINATE ASSEMBLY

Legal Program Q

Q: PROCEDURE;
   DECLARE LX LABEL;
   L: I = I+IA*IB-IC;
      LX = L;
      GO TO CHECK;
   M: I = I+1;
      LX = M;
   CHECK: IF I<LIMIT THEN GO TO LX;
END Q;

Translation for Q

* ASSEMBLER LANGUAGE PROGRAM FOR Q
*
* BALR 15,0      *SET REGISTER 15 AS BASE REGISTER
Q USING #,15      *INFORM ASSEMBLER OF USE OF R15 AS BASE REGISTER
L EQU *            *SYMBOLIC ADDRESS FOR PL/I LABEL L
   L 1,1            *LOAD I
   L 3,IA           *LOAD IA
   M 3-1,IB         *MULTIPLY BY IB
   AR 1,3           *ADD REGISTERS
   S 1,IC           *SUBTRACT IC
   ST 1,1            *STORE RESULT IN I
   MVC LX,=A(L)     *SET LX EQUAL TO ADDRESS OF L
   B CHECK          *BRANCH TO CHECK
   M EQU *            *SYMBOLIC ADDRESS FOR PL/I LABEL M
   L 1,1            *LOAD I
   A 1,'F'1'        *ADD =F'1'
   ST 1,1            *STORE RESULT IN I
   MVC LX,=A(M)     *SET LX EQUAL TO ADDRESS OF M
   CHECK EQU *        *SYMBOLIC ADDRESS FOR PL/I LABEL CHECK
   L 1,1            *LOAD I
   C 1,LIMIT        *COMPARE WITH LIMIT
   BNL Z            *BRANCH IF NOT LOW TO Z
   L 1,LX           *LOAD ADDRESS STORED IN LX
   BR 1              *BRANCH TO THIS ADDRESS
   Z EQU *            *SYMBOLIC ADDRESS OF Z
   SVC 0             *RETURN TO SUPERVISOR
   *
   LX DS F           *STORAGE FOR LX
   I DS F            *STORAGE FOR I
   IA DS F           *STORAGE FOR IA
   IB DS F           *STORAGE FOR IB
   IC DS F           *STORAGE FOR IC
   LIMIT DS F        *STORAGE FOR LIMIT
   END Q             *TERMINATE ASSEMBLY
Appendix 4 Canonical System Specification of Syntax of Little PL/I

1. letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

2. digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3. identifier ::= [letter]{0,10} [digit]{0,10} letter |

4. unsigned integer ::= [digit]{1,10} [digit]{0,10} digit |

5. fix-pt variable ::= [digit]{0,10} [digit]{0,10} [digit]{0,10} [digit]{0,10} [digit]{0,10} [digit]{0,10} [digit]{0,10} [digit]{0,10} [digit]{0,10} [digit]{0,10} 

6. label ::= identifier

7. label var ::= identifier

8. list ::= A list

9. 1. list ::= identifier |

10. disjoint ::= A disjoint

11. 1. disjoint ::= identifier |

12. boolean expression ::= IF boolean expression THEN non-DECLARE statement

13. label assign statement ::= label variable = <label> |

14. arith op ::= + | - | / |

15. arith exp ::= (primary) | arith exp arith op arith exp |

16. arith assign statement ::= fix-pt variable = arith exp |

17. DECLARE statement ::= DECLARE label variable LABEL: |

18. non-DECLARE statement ::= GO TO <label> |

19. statement ::= <statement> |

20. statement sequence ::= <statement> |

21. PL/I program ::= label: PROCEDURE; <statement sequence> 

END <label> ;
8.1. primary with fix-pt var
8.2.
9.1. goto stm with ref label-label var
9.2.
10. relop
11. boolean exp with fix-pt vars
12. if stm with ref label-label vars-fix pt vars
13.1. label assign stm with ref label-label var
13.2.
14. arith op
15.1. arith exp with fix-pt vars
15.2.
16. arith assign stm with fix-pt vars
17. declare stm with decl label var
18.1. non-declare stm with ref label-label vars-fix pt vars
18.2.
18.3.
18.4.
19.1. stm seq with stm labels-ref labels-label vars-decl label vars-fix pt vars
19.2.
19.3.
19.4.
20.1. 
20.2. legal stm seq with stm labels-vars
21. PL/I program
Appendix 6: Derivation of Example Legal Program

The following table specifies a derivation of the syntactically legal program P given in Appendix 2.

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>A derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 2</td>
<td>d derivation // list of premises // conclusion // n cannot // amid d // d_e derivation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>List of premises</th>
<th>Canon from App.4</th>
<th>Conclusion added to derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.1</td>
<td>I first letter for fix-pt var</td>
</tr>
<tr>
<td>2</td>
<td>1.1</td>
<td>A letter</td>
</tr>
<tr>
<td>3</td>
<td>5.2</td>
<td>A fix-pt var</td>
</tr>
<tr>
<td>4</td>
<td>8.1</td>
<td>A primary with fix-pt var A</td>
</tr>
<tr>
<td>5</td>
<td>15.1</td>
<td>A arith exp with fix-pt var A</td>
</tr>
<tr>
<td>6</td>
<td>14.</td>
<td>A digit</td>
</tr>
<tr>
<td>7</td>
<td>15.2</td>
<td>A arith exp with fix-pt var A</td>
</tr>
<tr>
<td>8</td>
<td>16.</td>
<td>A label</td>
</tr>
<tr>
<td>9</td>
<td>19.4</td>
<td>A non-declare stat with ref label A, label var A, fix-pt var A</td>
</tr>
<tr>
<td>10</td>
<td>1.</td>
<td>A letter</td>
</tr>
<tr>
<td>11</td>
<td>6.</td>
<td>A identifier</td>
</tr>
<tr>
<td>12</td>
<td>17.</td>
<td>A label</td>
</tr>
<tr>
<td>13</td>
<td>19.2</td>
<td>A: I = I+1; atm seq with int labels A ref labels A label var A deel label var A fix-pt var A,</td>
</tr>
<tr>
<td>14</td>
<td>18.</td>
<td>A digit</td>
</tr>
<tr>
<td>15</td>
<td>19.</td>
<td>A unsigned integer</td>
</tr>
<tr>
<td>16</td>
<td>20.</td>
<td>A primary with fix-pt var A</td>
</tr>
<tr>
<td>17</td>
<td>10.</td>
<td>A rel op</td>
</tr>
<tr>
<td>18</td>
<td>12.</td>
<td>A: I = I+1; atm seq with int labels A ref labels A label var A deel label var A fix-pt var A</td>
</tr>
<tr>
<td>19</td>
<td>22.</td>
<td>A: I = I+1; atm seq with int labels A ref labels A label var A deel label var A fix-pt var A</td>
</tr>
<tr>
<td>20</td>
<td>23.</td>
<td>A: I = I+1; atm seq with int labels A ref labels A label var A deel label var A fix-pt var A</td>
</tr>
<tr>
<td>21</td>
<td>24.</td>
<td>A: I = I+1; atm seq with int labels A ref labels A label var A deel label var A fix-pt var A</td>
</tr>
<tr>
<td>22</td>
<td>25.</td>
<td>A: I = I+1; atm seq with int labels A ref labels A label var A deel label var A fix-pt var A</td>
</tr>
<tr>
<td>23</td>
<td>26.</td>
<td>A: I = I+1; atm seq with int labels A ref labels A label var A deel label var A fix-pt var A</td>
</tr>
</tbody>
</table>

Derivation for example program = 7_9^1 = 6_9^2 = 1_9^1
We will assume that the string

from the derivation of Appendix 6 is also a member of the set named "derivations" for the derivation of this Appendix.