A formal system for the specification of the syntax and translation of computer languages*

by JOHN J. DONOVAN and HENRY F. LEDGARD
Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

This paper presents two basic results: the use of established methods of recursive definition to present a single method to

1. specify the syntax of computer languages (including context-sensitive requirements, such as the restrictions implied by declaration statements),
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 (including declaration, arithmetic, and conditional statements) and its translation into IBM System/260 assembler language.

Background

Our objective is to present a single method for expressing the syntax and translation of computer languages. The method was presented and first applied to specify only the syntax of computer languages in an earlier work by Donovan.4

The objective to develop methods for specifying either the syntax or 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 define and 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.9-14 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.25-28 However, most of the methods for specifying syntax have been shown to be equivalent to context-free phrase structure grammars and hence inadequate 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. In a Fortran program, for instance, the statement "GO TO 20" is not legal unless "20" occurs as a statement label in the same program. Restrictions like these 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 compilers27,28 and programming languages for expressing string transformations.20,21 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 the semantics of computer languages have also been made.16-19

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 syntactically 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 is 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. The power of the method of recursive definition is discussed and an ordered set of appendices is 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, and 5) a complete specification of the syntax of the subset and its translation into System/360 assembler language using the method of recursive definition.

Basis of formalization

The formalization for the method presented here evolved from Post’s canonical systems,1 and hence will be called canonic systems. Smullyan2 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 More3 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.4

Canonic systems were later applied by Ledgard5 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.3 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 or translation that a machine can perform. This confidence is a direct consequence of the works by Turing6,7 and Kleene.8-10 Here the notion of “recursive sets” (which are encompassed by canonic systems) was shown equivalent11-13 to the notion of functions computable by a “Turing machine”, and functions computable by a Turing machine were asserted14 to comprise every function or algorithm that is intuitively computable by machine.

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 canonical. A canon generally has the form

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

which may be interpreted informally:

If “a1” is a member of the set named “set A1”, and “a2” is a member of the set named “set A2”, . . . , and “an” is a member of the set named “set An”, then we can assert that “b” is a member of the set named “set B”.

The “a1” and “b” represent symbols from the finite alphabet; the “set A1” 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+1AxIB-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.

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

\[ \vdash A \text{ letter} \quad (1) \]
\[ \vdash B \text{ letter} \quad (2) \]
\[ \vdots \]
\[ \vdash Z \text{ letter} \quad (26) \]

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”:

\[ L_1 \text{ letter} \vdash L_1 \text{ identifier} \quad (27) \]
\[ L_1 \text{ letter} \vdash L_2 \text{ letter} \vdash L_1L_2 \text{ identifier} \quad (28) \]
\[ \vdots \]
\[ L_1 \text{ letter} \vdash L_2 \text{ letter} \vdash \ldots \vdash L_8 \text{ letter} \]
\[ \vdash L_1L_2L_1L_2L_1L_2L_1L_2 \text{ identifier} \quad (34) \]

These canons may be read:

If “L_1” is a member of the set named “letter”, then we can assert that “L_1” is a member of the set named “identifier”.

If “L_1” is a member of the set named “letter” and “L_2” is a member of the set named “letter”, then we can assert that “L_1L_2” is a member of the set named “identifier”.

And so on.

The canons* specify a set consisting of identifiers of one to eight capital English letters. The lower case (possibly subscripted or superscripted) English letters \( L_1 \) through \( L_8 \) are variables that represent members of the set named “letter”. The terms \( L_1 \), “L_1L_2” are terms. A term is a string of variables or symbols from the object alphabet (e.g. “L_1”, “L_1L_2”, or “A_1A_2”). The sign “\( \vdash \)” is the conjunction sign. The strings “L_1 letter”, “L_2 letter”, “L_1 letter” are premises. The “\( \vdash \)” before an “\( \vdash \)” separates premises, all of which must be satisfied to assert the conclusion. The strings “L_1 letter”, “L_1 identifier”, “L_2 letter”, “L_1L_2 identifier” are remarks. A remark is either a premise or a conclusion.

*Actually, these canons should properly be called canon schema, which denote instances of canons.
The vocabulary used to describe a canon is summarized as follows:

- letter \[\rightarrow\] identifier
- variable \[\rightarrow\] variable
- term predicate \[\rightarrow\] term predicate
- premise \[\rightarrow\] conclusion
- remark \[\rightarrow\] remark

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:

\[\begin{align*}
A\text{list} &\rightarrow i \text{ identifier} \rightarrow i, \text{ list} \\
A\text{list} &\rightarrow a \text{ list} \rightarrow ab \text{ list}
\end{align*}\]

(35) (36) (37)

We can use canons (1) through (37) to derive the conclusion 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 that "A" is a member of the set named "identifier." 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)

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

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 \(C_1, C_2, \ldots, C_n\) are conclusions with identical premises \(Q\), the canons
   \[Q \vdash C_1 \quad Q \vdash C_2 \quad \ldots \quad Q \vdash C_n\]
   may be abbreviated
   \[Q \vdash C_1 \quad C_2 \quad \ldots \quad C_n\]

2. If \(t_1, t_2, \ldots, t_n\) are terms denoting members of the same set named "S," the remarks
   \[t_1S \quad t_2S \quad \ldots \quad t_nS\]
   may be abbreviated
   \[t_1S \quad \ldots \quad t_nS\]

Thus the canons for "letter" may be abbreviated with abbreviation 1:

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

or further abbreviated with abbreviation 2:

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

The canons for "identifier" may be abbreviated*

\[A\|B\| \ldots \| Z \text{ letter} \rightarrow A\|B\| \ldots \| Z \text{ identifier}\]

This canon may be read:

If "A," "B," ..., and "Z" are members of the set named "letter," then we can assert that "A," "B," ..., and "Z" are members of the set named "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. 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 \(n > 1\), i.e., predicates of degree \(n, n > 1\). We use the notation

\[<a_1, a_2, \ldots, a_n>\]

where \(a_1, a_2, \ldots, a_n\) are terms, to denote an ordered n-tuple. The sign "<" separates elements of an ordered 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 second element may be abbreviated

\[Q \vdash C_1 \quad C_2 \quad \ldots \quad C_n\]

*To obtain the given abbreviation for "identifier," we must first change the premises of each of the eight unabbreviated canons to "A letter \(\vdash A, letter \quad \ldots \quad A, letter." This change does not effect the validity of the canons.
The set named "in" is defined:
\[
\begin{align*}
\text{abc list} & \vdash <b,abc> \text{ in } (38) \\
<a_1..a_n> & \vdash <b,a> \text{ in } <a_1..a_n> \text{ in } (39)
\end{align*}
\]
To suggest more strongly the relationship among the elements of an n-tuple, we allow premises and conclusions of the form
\[
\begin{align*}
\langle a_1..a_n \rangle & \text{ text1 text2..textn }^{-1} \\
\end{align*}
\]
to be written
\[
\begin{align*}
a_1 \text{ text1 } a_2 \text{ text2..textn }^{-1} \ a_n
\end{align*}
\]
Thus the canons for "in" become
\[
\begin{align*}
\text{abc list} & \vdash b \text{ in abc } (38) \\
a & \vdash b \text{ in } \epsilon \dashv \text{ ab in } \epsilon \ (39)
\end{align*}
\]
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 "\epsilon" and the list "b" is contained in the list "\epsilon", then we can assert that the list "ab" is contained in the list "\epsilon".

The following conclusions can be derived from canons (1) through (39):
\[
\begin{align*}
A, & \text{ in A,B, } A_1, \text{ in } A, \ R,S, \text{ in S,T,R,} \\
\text{ALPHA}, & \text{ in } \text{ALPHA,BETA.}
\end{align*}
\]
Finally, we extend the use of the symbol "\text{\psi}\" to abbreviate the writing of n-tuples by allowing remarks of the form
\[
\begin{align*}
a_1 \text{ text1 } a_2 \text{ text2..textn }^{-1} \ a_n
\end{align*}
\]
to be abbreviated
\[
\begin{align*}
a_1a_1' \text{ text1 a_2a_2' text2..textn }^{-1} \ a_na_n'
\end{align*}
\]
Thus canon (39)
\[
\begin{align*}
a & \vdash b \text{ in } \epsilon \dashv \text{ ab in } \epsilon
\end{align*}
\]
may be abbreviated
\[
\begin{align*}
\text{ab in } \epsilon \vdash \text{ ab in } \epsilon
\end{align*}
\]

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 1-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 must be mutually disjoint, we define the set named "differ" (canons 7.7 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,>" is a member of the set named "\text{disjoint}". To define the set named "\text{disjoint}", we define a set named "\text{differ}" (canons 7.7 through 7.9). Members of the set named "\text{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:
\[
\begin{align*}
\text{label } & \vdash \text{GO TO } \lambda; \text{ goto stm with ref label } \lambda, \\
\text{label var } & \lambda
\end{align*}
\]
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, and the third element the label variable for the GO TO statement. The canon has the following instance:

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

or

A label \( \uparrow \) <GO TO A; A, A> goto stm with ref label-label var

Likewise, with every arithmetic assignment statement we keep track of the list of its fixed-point variables (to check later if the list of fixed-point variables for a program is disjoint from the lists of statement labels, declared label variables, and procedure label for the program). Hence the canons for an arithmetic expression and an arithmetic assignment statement specify sets of ordered pairs, in which the second element of each pair is the list of associated fixed-point variables. In the same manner, we define a set of ordered 4-tuples to specify an IF statement, a set of ordered pairs for a DECLARE statement.

Canons 19.1 through 19.4 specify a statement sequence consisting of a single (possibly labeled*) statement. Canon 20.1

\[
\text{s} \text{stm seq with stm labels } \mathcal{L}, \mathcal{L}', \text{ ref labels } \mathcal{L}, \mathcal{L}', \text{ label vars } \mathcal{L}, \mathcal{L}', \text{ decl label vars } \mathcal{L}, \mathcal{L}', \text{ fix-pt vars } \mathcal{L}, \mathcal{L}', \text{ \text{s legal stm seq with stm labels } } \mathcal{L}, \mathcal{L}', \text{ ref labels } \mathcal{L}, \mathcal{L}', \text{ label vars } \mathcal{L}, \mathcal{L}', \text{ decl label vars } \mathcal{L}, \mathcal{L}', \text{ fix-pt vars } \mathcal{L}, \mathcal{L}', \text{ (20.1)}
\]

\[
\uparrow \text{s} \text{ st seq with stm labels } \mathcal{L}, \mathcal{L}', \text{ ref labels } \mathcal{L}, \mathcal{L}', \text{ label vars } \mathcal{L}, \mathcal{L}', \text{ decl label vars } \mathcal{L}, \mathcal{L}', \text{ fix-pt vars } \mathcal{L}, \mathcal{L}', \text{ label vars } \mathcal{L}, \mathcal{L}', \text{ decl label vars } \mathcal{L}, \mathcal{L}', \text{ fix-pt vars } \mathcal{L}, \mathcal{L}', \text{ (20.1)}
\]

The premise "\( \mathcal{L}, \text{ disjoint } \mathcal{L}' \)" insures that the statement labels for each statement sequence are different.

Canon 20.2

\[
\text{s} \text{ st seq with stm labels } \mathcal{L}, \text{ ref labels } \mathcal{L}, \text{ label vars } \mathcal{L}, \text{ decl label vars } \mathcal{L}, \text{ fix-pt vars } v \vdash v \text{ disjoint } \mathcal{L}, \mathcal{L}', v \text{ disjoint } \mathcal{L}, \mathcal{L}', \text{ in } v \text{ in } \mathcal{L}, \mathcal{L}', \text{ in } \mathcal{L}, \mathcal{L}', \text{ in } \mathcal{L}, \mathcal{L}', \text{ (20.1)}
\]

\[
\uparrow \text{s legal st seq with stm labels } \mathcal{L}, \mathcal{L}', \text{ vars } \mathcal{L}, \mathcal{L}', \text{ (20.1)}
\]

specifies a syntactically legal statement sequence. The premises "\( v \text{ disjoint } \mathcal{L}, \mathcal{L}' \)" and "\( \mathcal{L}, \text{ disjoint } \mathcal{L}, \mathcal{L}' \)" 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 "\( \mathcal{L}, \text{ disjoint } \mathcal{L}, \mathcal{L}' \)" and "\( \mathcal{L}, \text{ disjoint } \mathcal{L}, \mathcal{L}' \)" 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.

\[
\text{a label } \uparrow \text{s legal st seq with stm labels } \mathcal{L}, \mathcal{L}', \text{ vars } v \vdash v \text{ disjoint } \mathcal{L}, \mathcal{L}', \text{ in } \mathcal{L}, \mathcal{L}', \text{ in } \mathcal{L}, \mathcal{L}', \text{ PL/I program.}
\]

specifies a syntactically legal PL/I program. The premise "\( \mathcal{L}, \text{ disjoint } \mathcal{L}, \mathcal{L}' \)" insures that the procedure label "\( \mathcal{L} \)" 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.

**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 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 indicate further applications of canonic systems and conclude with a discussion of the use of canonic systems as a method of definition.

Consider first canon 9.1 of Appendix 4:

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

(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 A *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 “*BRANCH TO A” is a comments entry. We may specify this translation by modifying canon 9.1:

\[ \text{label} \rightarrow \text{GO TO } \ell; \text{goto stm with ref label } \ell, \text{ label varAssembler stms} \]  

B A *BRANCH TO A

(9.1')

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 statement for the GO TO statement.

The instance of canon 9.1’ for the GO TO statement “GO TO A;” is simply

A label \rightarrow \text{GO TO A; goto stm with ref label A,} \text{label varAssembler stms}  

B A *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 in canon 9.2'.

The assembler language statements for an IF statement, such as

IF I<5 THEN <non-DECLARE statement>, might be as follows:

L 1,1 *LOAD I  
C 1,=F'5' *COMPARE WITH =F'5'  
BNL Z *BRANCH IF NOT LOW TO Z

(translation for non-DECLARE statement  
following the boolean expression)

Z EQU *  

*SYMBOLIC ADDRESS OF Z

For these 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

*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.
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 use (canon 12.') in specifying the EQU assembler statement needed to terminate the assembler statements for the IF statement.

Canons 10.', 11.', and 12.', in conjunction with canons 8.1' and 8.2', specify the translation of an IF statement according to this format. The premise "\( \not \exists \text{ disjoint } \ell_s \)" in canon 12.' 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 label needed to specify the proper transfer of control for a false boolean expression may later be specified to be different from the other assembler labels and PL/I identifiers.

Canons 13.1 and 13.2 of Appendix 4., the canons for a label assignment statement, are easily modified to specify their translation in canons 13.1' and 13.2'. We next consider canons 14., 15., and 16., of Appendix 4., the canons for an arithmetic expression:

\[
\begin{align*}
\{ & +, -, \ast, \div \} \text{ arith op} & \quad (14) \\
\{ & \text{ primary with fix-pt vars } v \} \quad (15) \\
\text{ with fix-pt vars } v & \text{ arith exp} \quad \text{ with fix-pt vars } v \\
\text{ with fix-pt vars } v & \quad \text{ arith exp} \quad \text{ with fix-pt vars } vv' \\
\end{align*}
\]

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 (+, -, \ast, and \div) into two classes, one for the addition operators, + and -, and one for the multiplication operators, \ast and \div, (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, \ast and \div,
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+I" as a legal arithmetic expression with assembler statements:

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

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

\[
\begin{align*}
L & 1,A \quad *\text{LOAD } A \quad \text{(canon 15.1')}
L & 3,B \quad *\text{LOAD } B \quad \text{(canon 15.2')}
M & 2,C \quad *\text{MULTIPLY BY } C
M & 3-1,D \quad *\text{MULTIPLY BY } D \quad \text{(canon 15.3')}
AR & 1,3 \quad *\text{ADD REGISTERS} \quad \text{(canon 15.6')}
\end{align*}
\]

The remaining canons of Appendix 5. readily follow. Canon 16.', specifies the assembler store instruction appended to the assembler statements for an arithmetic expression. Canon 17.', is identical to canon 17. of Appendix 4. Canons 18.1' through 20.2', the canons for a non-DECLARE statement and a statement sequence, are similar to those of Appendix 4., with the added predicate elements needed to carry the associated assembler labels and assembler statements. The premise "\( \not \exists \text{ disjoint } \ell_s \)" added to canon 20.1' insures that the assembler labels differ from each other. The premise "\( \not \exists \text{ disjoint } \ell_s \) \& \( \not \exists \text{ disjoint } L_v \)" 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' 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":

\[
\begin{align*}
\{ & \text{ A set of vars with assembler data stmsA } \\
\text{ A set of vars with assembler data stmsA } & \quad \downarrow 1 \text{ identifier} \\
\downarrow 1 \text{, disjoint } \downarrow I, \text{ set of vars with assembler data stms } & \\
I & \quad \text{DS } F \quad *\text{STORAGE FOR } I
\end{align*}
\]

Finally, canon 21., of Appendix 4., the canon for a syntactically legal little PL/I program, is modified to give canon 21.3'
A formal system for specification of syntax and translation of computer languages

The added premise “v‘ set of vars with assembler data stms s_d‘ v in v‘” 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 canonical 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 canonical systems can be used to write one specification for both the syntax of a source language and its translation into a target language.

Related applications

An immediate use of canonical systems is in the development of a generalized translator, i.e., a translator that is independent of both source and target languages. We have used canonical systems to define a set by specifying canons for generating its members. To use a canonical system specification of the syntax and translation of a language as a data base for a generalized translator, an algorithm to recognize source language strings specified by a canonical system and construct their associated translation is needed. No algorithm for recognizing and translating strings specified by a canonical system is presented in this paper. However, work on the application of canonical systems to a generalized translator has been initiated and is more fully described elsewhere. (6, 7, 8)

Canonic system may be used to obtain structural information (e.g., syntactic trees) of programs. The series of canons used in generating a string provides structural information about the generated string. This series of canons constitute a “derivation” of the generated string. Only an intuitive notion of a derivation has been indicated here. However, using the concept of higher level canonical systems, the rules for constructing a derivation can be formalized, and, in turn, the rules for constructing a structural description of a derived string made explicit. Higher level canonical systems, and their application to provide structural descriptions, are also described elsewhere. (6, 7)

DISCUSSION

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 canonical 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 canonical systems to specify both the syntax and translation of these features.

However, an equally important issue is whether canonical systems provides a natural and concise specification of these features. A canonical system for the complete syntax of only one language, the simulation of the complete syntax of one language, the simulation language GPSS, (4) has been written.

It is clear that a complete specification of the syntax of PL/I and its translation to assembler language might be unduly large. We feel that this largeness is due to three principal factors. First, the PL/I and System/360 assembler languages are complicated, and the degree of the canonical system predicates grows as the complexity of the source or target language grows. 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 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 given in text 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. Present research has yielded a notational convention that will eliminate much of the detail of Appendices 4 and 5. We hope that this modification to the notation for canonic systems will make canonic systems as suitable as Backus-Naur Form for describing programming languages to people. However, in their present form canonic systems, although yielding larger specifications, provide complete descriptions in a precise language that a machine can be instructed to understand with present techniques. 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 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 "GO (a label & goto stm with ref label & 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 interterminal 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 would be desirable to use a target language other than an assembler language. This new target language would have a set of primitives in which the constructions in a large class of languages could readily be expressed. This target language has not been developed.

REFERENCES

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

1 EMIL L POST
Formal reductions of the general combinatorial decision problem
Am J Math 65 pp 197-217 1964 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 RAYMOND M SMULLYAN
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 TRENCHARD MORE

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.
32 and applied to the study of various mathematical systems.

4 JOHN J DONOVAN

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 HENRY F LEDGARD

A scheme for the translation of computer languages
Ph. D. thesis proposal MIT Cambridge Mass July 1967. Here canonic systems were applied to specify the translation of computer languages.

The following references 1) describe canonic systems and their use in specifying syntax and translation, and 2) present higher level canonic systems, their use in formalizing the notion of a derivation of a program string, and a discussion of the application of canonic systems to a generalized translator for computer languages.

6 JOHN J DONOVAN and HENRY F LEDGARD

Canonic systems and their application to programming languages
Project MAC memorandum MAC-M-347 MIT Cambridge Mass April 1967

7 JOHN J DONOVAN and HENRY F LEDGARD

Canonic systems and their application to computer languages
Submitted paper August 1967.

The following references 1) describe canonic systems and their use in specifying syntax and translation, and 2) present higher level strings specified by a canonic system and generate their translation.

8 JOSEPH W ALSOP

A canonic translator

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

9 J W BACKUS

The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference
Proc Inter'l conf information processing UNESCO Paris pp 125-132 June 1959

10 D E KNUTH

Backus-normal form vs Backus-Naur form
Comm ACM vol 7 p 735 December 1964

11 W H BURKHARDT

Meta-language and Syntax specification
Comm ACM vol 8 p 304-305 May 1965

12 KEIVERSON

A method of syntax specification
Comm ACM vol 7 p 588-589 October 1964

13 R W FLOYD

The syntax of programming languages—a survey
IEEE Trans of electronic computers pp 346-353 August 1964

14 T B STEEL Editor

Formal language description languages for computer programming
North-Holland Publishing Co Amsterdam The Netherlands 1966

15 JOHN MccARTHY

Towards a mathematical science of computation

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

16 PETER LINDIN

The calculus approach
Advances in programming and non-numerical computation pergamon press 1966

17 A VAN WIJN-AARDEN

Recursive definition of syntax and semantics
IFIP working conf Baden September 1964

18 J A FELDMAN

A formal semantics for computer languages and its application in a compiler-compiler
Comm ACM vol 9 pp 3-9 January 1966

19 N WIRTH and H WEBER

EULER a generalization of ALGOL and its formal definition

The following papers describe programming languages for expressing string transformations.

20 D J FARBER R E GRISWOLD and I P POLONSKY

SNOBOL a string manipulation language
Journal of the ACM vol 11 no 2 pp 21-30 January 1966

21 CALVIN N MOOERS

TRAC a procedure-describing language for the reactive typewriter
Comm of the ACM vol 9 no 3 pp 215-219 March 1966

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

22 JOSEPH WEIZENBAUM

ELIZA—A computer program for the study of natural language communication between man and machine
Comm of the ACM vol 9 no 1 pp 36-45 January 1966

The following references and ref 13 point out inadequacies of past specifications in specifying the syntax of programming languages.

23 P GILBERT

On the syntax of algorithmic languages
J ACM vol 13 pp 90-107 January 1966

24 A C DIFORINO

Some remarks on the syntax of symbolic programming languages
Comm ACM vol 6 pp 456-460 August 1963

The following references and (ref. 18) describe generalized translators that have been implemented as table-driven compilers.

25 E T IRONS

A syntax directed compiler for ALGOL 60
Comm ACM vol 4 pp 51-55 January 1961

26 R A BROOKER and D MORRIS

A general translation program for Phrase-structure languages
J ACM vol 9 pp 1-10 January 1962

27 T E CHEATAM JR and K SATTLEY

Syntax-directed compiling
Proc spring joint computer conf Spartan Books Baltimore Md vol 25 pp 31-57 1964

28 C L LIU G D CHANG and R E MARKS

The design and implementation of a table driven compiler
AFIPS vol 30 1967 Spring Joint Computer conference Spartan Books Washington 1967 A detailed discussion of this translator is given in a project MAC technical report MAC-TR-42
The following references describe to an extent the syntax of PL/I.

1. H. C. Chladek, V. Kudielka, and E. Nuehold
   "Syntax of PL/I"
   IBM Technical report TR 25.058
   Vienna Laboratory September 1965

2. Formal definition of PL/I
   IBM Technical Report TR 25.071
   PL/I definition group of the Vienna Laboratory
   December 1966

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

Paul C. Rosenbloom
"The elements of mathematical logic"
Dover publications Inc 1780 Broadway New York 19 N Y 1950

Trenchard More
"Introduction to mathematical logic"
Princeton New Jersey 1956 vol 1

Alonzo Church
"Introduction to mathematical logic"
Princeton New Jersey 1956 vol 1

A. M. Turing
"Computability and lambda definability"
J. sym. logic vol 2 pp 153-163 1937

Stephen C. Kleene
"Lambda-definability and recursiveness"
Duke math. j vol 2 pp 340-353 1936

Noam Chomsky
"On certain formal properties of grammars"
in readings of mathematical psychology
Luce R. D. et al.
John Wiley and Sons Inc New York N Y vol 2 pp 125-155 1965

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

1. IBM System/360 operating system PL/I language specification
   IBM systems reference library form C28-6571-3

2. A programmer's introduction to the IBM system/360
   architecture instructions and assembler language
   Student text form C28-1646-1

3. IBM system/360 operating system assembler language
   IBM system reference library form C28-6514-4

Appendix: Summary of Notation for Canon Systems

To provide the reader with a concise reference, the canonical system notation used in this paper will be presented by a series of short examples:

### Basic Constructions:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Interpretation</th>
<th>New Construction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A letter</td>
<td>&quot;A&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>B letter</td>
<td>&quot;B&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>C letter</td>
<td>&quot;C&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>f letter</td>
<td>&quot;f&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>A identifier</td>
<td>&quot;A&quot; is a member of the set named &quot;identifier&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>B identifier</td>
<td>&quot;B&quot; is a member of the set named &quot;identifier&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>C identifier</td>
<td>&quot;C&quot; is a member of the set named &quot;identifier&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>D identifier</td>
<td>&quot;D&quot; is a member of the set named &quot;identifier&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>A, B, C, D differ</td>
<td>The ordered pair &quot;&lt;A, B, C, D&gt;&quot; is a member of the set named &quot;differ&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
</tbody>
</table>

### Abbreviations and Alternate Notations:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Interpretation</th>
<th>New Construction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A letter</td>
<td>&quot;A&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>B letter</td>
<td>&quot;B&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>A letter</td>
<td>&quot;A&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>B letter</td>
<td>&quot;B&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>A letter</td>
<td>&quot;A&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>B letter</td>
<td>&quot;B&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>A letter</td>
<td>&quot;A&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
<tr>
<td>B letter</td>
<td>&quot;B&quot; is a member of the set named &quot;letter&quot;.</td>
<td><img src="image" alt="New Construction" /></td>
</tr>
</tbody>
</table>

---

From the collection of the Computer History Museum (www.computerhistory.org)
### Appendix 2: Two syntactically legal programs in Little PL/I and their translation into IBM System/360 Assembler Language

#### Legal Program P

**PROEDURE:**

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

#### Translation for P

```
* * * ASSEMBLER LANGUAGE PROGRAM FOR P *

* * * * SET REGISTER 15 AS BASE REGISTER *
BALR 15,0

* * * INFORM ASSEMBLER THAT R15 IS BASE REG *
USING *,15

* * * SYMBOLIC ADDRESS FOR PL/I LABEL A *
EQU *

L I, I
A 1, 1
ST 1, 1
L 1, 1
C 1, 1
BNL Z
B A
Z EQU *

* * * SYMBOLIC ADDRESS OF Z *
SVC 0

* * * STORAGE FOR I *
I

* * * TERMINATE ASSEMBLY *
END P
```

#### Legal Program Q

**PROEDURE:**

```
Q: DECLARE LX LABEL;
    L: I = I + 1;
        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 *

* * * SET REGISTER 15 AS BASE REGISTER *
BALR 15,0

* * * INFORM ASSEMBLER THAT R15 IS BASE REG *
USING *,15

* * * SYMBOLIC ADDRESS FOR PL/I LABEL L *
EQU *

L I, I
L 1, 1
ST 1, 1
M 1, 1
AR 1, 3
S 1, 1
ST 1, 1
MVC LX, =A(L)
B CHECK

* * * SYMBOLIC ADDRESS OF L *
EQU *

L I, I
A 1, 1
ST 1, 1
MVC LX, =A(M)

* * * STORAGE FOR I *
I

* * * STORAGE FOR LIMIT *
IA

* * * STORAGE FOR IA *
IB

* * * STORAGE FOR IB *
IC

* * * STORAGE FOR IC *
LIMIT

* * * STORAGE FOR LIMIT *
LX

* * * TERMINATE ASSEMBLY *
END Q
```
### Appendix 3 BACKUS-NAUR FORM SPECIFICATION OF SYNTAX OF LITTLE PL/I

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>&lt;letter&gt;</code> ::= A</td>
</tr>
<tr>
<td>2.</td>
<td><code>&lt;digit&gt;</code> ::= 0</td>
</tr>
<tr>
<td>3.</td>
<td><code>&lt;identifier&gt;</code> ::= [letter]</td>
</tr>
<tr>
<td>4.</td>
<td><code>&lt;unsigned integer&gt;</code> ::= [0-9][1-9] unsigned integer</td>
</tr>
<tr>
<td>5.</td>
<td><code>&lt;fix-pt variable&gt;</code> ::= [][0-9][1-9][1-9] [letter]</td>
</tr>
<tr>
<td>6.</td>
<td><code>&lt;label&gt;</code> ::= &lt;identifier&gt;</td>
</tr>
<tr>
<td>7.</td>
<td><code>&lt;label variable&gt;</code> ::= &lt;identifier&gt;</td>
</tr>
<tr>
<td>8.</td>
<td><code>&lt;primary&gt;</code> ::= &lt;unsigned integer&gt;</td>
</tr>
<tr>
<td>9.</td>
<td><code>&lt;GO TO statement&gt;</code> ::= GO TO &lt;label&gt;</td>
</tr>
<tr>
<td>10.</td>
<td><code>&lt;rel op&gt;</code> ::= &lt;rel op&gt;</td>
</tr>
<tr>
<td>11.</td>
<td><code>&lt;boolean expression&gt;</code> ::= &lt;rel op&gt;</td>
</tr>
<tr>
<td>12.</td>
<td><code>&lt;IF statement&gt;</code> ::= IF &lt;boolean expression&gt;</td>
</tr>
<tr>
<td>13.</td>
<td><code>&lt;label assign statement&gt;</code> ::= &lt;label variable&gt;=&lt;label&gt;</td>
</tr>
<tr>
<td>14.</td>
<td><code>&lt;arithmetic op&gt;</code> ::= + - / *</td>
</tr>
<tr>
<td>15.</td>
<td><code>&lt;arithmetic exp&gt;</code> ::= &lt;primary&gt;&lt;arithmetic op&gt;</td>
</tr>
<tr>
<td>16.</td>
<td><code>&lt;arithmetic assign statement&gt;</code> ::= &lt;fix-pt variable&gt;=&lt;arithmetic exp&gt;</td>
</tr>
<tr>
<td>17.</td>
<td><code>&lt;DECLARE statement&gt;</code> ::= DECLARE &lt;label variable&gt;</td>
</tr>
<tr>
<td>18.</td>
<td><code>&lt;non-DECLARE statement&gt;</code> ::= GO TO &lt;label&gt;</td>
</tr>
<tr>
<td>19.</td>
<td><code>&lt;statement&gt;</code> ::= &lt;non-DECLARE statement&gt;</td>
</tr>
<tr>
<td>20.</td>
<td><code>&lt;statement sequence&gt;</code> ::= &lt;statement&gt; sequence</td>
</tr>
<tr>
<td>21.</td>
<td><code>&lt;PL/I program&gt;</code> ::= &lt;label&gt;: PROCEDURE;</td>
</tr>
</tbody>
</table>

**Notation:** In addition to pure Backus-Naur Form, we use the brackets [ ] to designate any number from k₁ through k₂ of occurrences of the enclosed expression.

### Appendix 4 CANONIC SYSTEM SPECIFICATION OF SYNTAX OF LITTLE PL/I

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>letter</code> ::= A</td>
</tr>
<tr>
<td>2.</td>
<td><code>digit</code> ::= 0</td>
</tr>
<tr>
<td>3.</td>
<td><code>identifier</code> ::= [letter]</td>
</tr>
<tr>
<td>4.</td>
<td><code>unsigned integer</code> ::= [0-9][1-9] unsigned integer</td>
</tr>
<tr>
<td>5.1</td>
<td><code>first letter for fix-pt var</code> ::= <code>letter</code></td>
</tr>
<tr>
<td>5.2</td>
<td><code>fix-pt var</code> ::= <code>letter</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>label</code> ::= &lt;identifier&gt;</td>
</tr>
<tr>
<td>7.1</td>
<td><code>label var</code> ::= &lt;identifier&gt;</td>
</tr>
<tr>
<td>7.2</td>
<td><code>list</code> ::= A list</td>
</tr>
<tr>
<td>7.3</td>
<td><code>identifier</code> ::= i, list</td>
</tr>
<tr>
<td>7.4</td>
<td><code>a b list</code> ::= ab list</td>
</tr>
<tr>
<td>7.5</td>
<td><code>a b c list</code> ::= b in abc</td>
</tr>
<tr>
<td>7.6</td>
<td><code>a b c d e f</code> ::= ab in f</td>
</tr>
<tr>
<td>7.7</td>
<td><code>differ</code> ::= A</td>
</tr>
<tr>
<td>7.8</td>
<td><code>z z z</code> differ A</td>
</tr>
<tr>
<td>7.9</td>
<td><code>x y letter</code> differ y y x and s y differ s</td>
</tr>
<tr>
<td>7.10</td>
<td><code>x y differ</code> y y differ x</td>
</tr>
<tr>
<td>7.11</td>
<td><code>A disjoint A</code></td>
</tr>
<tr>
<td>7.12</td>
<td><code>i disjoint i</code></td>
</tr>
<tr>
<td>7.13</td>
<td><code>x disjoint y</code> disjoint f f disjoint x y</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
Formal system for specification of syntax and translation of computer languages

Appendix 4 (continued) CANONIC SYSTEM SPECIFICATION OF LITTLE PL/I

<table>
<thead>
<tr>
<th>8.1</th>
<th>primary with fix-pt var</th>
<th>i unsigned integer</th>
<th>i primary with fix-pt var A</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2</td>
<td>v fix-pt var</td>
<td>v primary with fix-pt var v,</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>goto stmt with ref label-label var</td>
<td>f label</td>
<td>GO TO t; goto stmt with ref label t, label var A</td>
</tr>
<tr>
<td>9.2</td>
<td>f label var</td>
<td>GO TO t; goto stmt with ref label A label var t,</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>relop</td>
<td>t &lt; s = r &gt; rel op</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>boolean exp with fix-pt vars</td>
<td>p ± p' primary with fix-pt var v + v' ± r ± op ± p ± boolean exp with fix-pt vars vv'</td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>if stmt with ref label-label vars-fix-pt vars</td>
<td>b boolean exp with fix-pt vars v ± s non-declare stmt with ref labels s label vars s fix-pt vars v' ± IF b THEN s; if stmt with ref label A label vars A fix-pt vars vv'</td>
<td></td>
</tr>
<tr>
<td>13.1</td>
<td>label assign stmt with ref label-label vars</td>
<td>v label var ± f label</td>
<td>v = t; label assign stmt with ref label t, label var v,</td>
</tr>
<tr>
<td>13.2</td>
<td>± v' label var</td>
<td>v = v' ± label assign stmt with ref label A label vars v, v',</td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>arith op</td>
<td>t + + - + • ± arith op</td>
<td></td>
</tr>
<tr>
<td>15.1</td>
<td>arith exp with fix-pt vars</td>
<td>p primary with fix-pt var v ± p arith exp with fix-pt vars v</td>
<td></td>
</tr>
<tr>
<td>15.2</td>
<td>± v' arith exp with fix-pt vars v ± v' ± r ± arith op ± a ± arith exp with fix-pt vars vv'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>arith assign stmt with fix-pt vars</td>
<td>v ± v' if stmt with fix-pt vars v ± v' ± v ± a ± arith assign stmt with fix-pt vars vv',</td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>declare stmt with decl label var</td>
<td>v label var ± DECLARE v LABEL; declare stmt with decl label var v,</td>
<td></td>
</tr>
<tr>
<td>18.1</td>
<td>non-declare stmt with ref label-label vars-fix-pt vars</td>
<td>g goto stmt with ref label t, label var f,</td>
<td></td>
</tr>
<tr>
<td>18.2</td>
<td>± v' non-declare stmt with ref labels t label vars t fix-pt vars A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.3</td>
<td>i if stmt with ref label A label vars A fix-pt vars v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.4</td>
<td>± v' non-declare stmt with ref labels t label vars t fix-pt vars A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.1</td>
<td>stmt seq with stmt labels-ref label-label vars-decl label vars-fix-pt vars</td>
<td>v ± v' non-declare stmt with ref labels t label vars t fix-pt vars A non-declare stmt with ref labels t label vars t fix-pt vars A</td>
<td></td>
</tr>
<tr>
<td>19.2</td>
<td>± v' stmt seq with stmt labels A ref labels t label vars f decl label vars A fix-pt vars A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.3</td>
<td>d declare stmt with decl label var v ± f label</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.4</td>
<td>d declare stmt with decl label var v ± f label</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.1</td>
<td>s ± s' stmt seq with stmt labels t' ± f' s' ± f' ref labels t' ± f' label vars t' ± f' decl label vars A fix-pt vars A s ± s' stmt seq with stmt labels t' ± f' s' ± f' ref labels t' ± f' label vars t' ± f' decl label vars A fix-pt vars A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.2</td>
<td>legal stmt seq with stmt labels-vara</td>
<td>s ± s' stmt seq with stmt labels t' ± f' s' ± f' ref labels t' ± f' label vars t' ± f' decl label vars A fix-pt vars A</td>
<td></td>
</tr>
<tr>
<td>21.</td>
<td>PL/I program</td>
<td>t label</td>
<td>t = PROCEDURE; s END t; PL/I program</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
Appendix 5 CANONIC SYSTEM SPECIFICATION OF SYNTAX OF LITTLE PL/II
AND TRANSLATION INTO IBM SYSTEM/360 ASSEMBLER LANGUAGE
(Canons 1. through 7.13. same as canons 1. through 7.13. in Appendix 4.)

8.1. primary with fix-pt var-operand code
8.2. i unsigned integer

9.1. goto stmt with ref label-
label var-assembly stmt
9.2. i primary with fix-pt var

10.1. rel op with branch code-message
10.2. boolean exp with fix-pt var-
branch label assembly stmt

11.1. if stmt with ref label-
label var-fix-pt var-
assembly labels-assembly stmt
11.2. label assign stmt with ref label-
label var-assembly stmt

12.1. add op with op code-message
12.2. arith exp with fix-pt var-
result in reg-assembly stmt

13.1. term with fix-pt var-
result in reg-assembly stmt

14.1. arith exp with fix-pt var-
result in reg-assembly stmt

15.1. primary with fix-pt var-
operand code
15.2. i primary with fix-pt var

16.1. arith exp with fix-pt var-
result in reg-assembly stmt
16.2. a arith exp with fix-pt var-
result in reg-assembly stmt
16.3. a arith exp with fix-pt var-
result in reg-assembly stmt
16.4. a arith exp with fix-pt var-
result in reg-assembly stmt
16.5. a arith exp with fix-pt var-
result in reg-assembly stmt
16.6. a arith exp with fix-pt var-
result in reg-assembly stmt
Formal system for specification of syntax and translation of computer languages  

Appendix 5 (continued) CANONIC SYSTEM SPECIFICATION OF SYNTAX OF LITTLE PL/I  
AND TRANSLATION INTO IBM SYSTEM/360 ASSEMBLER LANGUAGE

16.1  arith assign stmt with fix-pt var a assembler stmt
  v1 fix-pt var a arith exp with fix-pt var a result in reg 1 assembler stmt
  s a arith assign stmt with fix-pt var a assembler stmt

17.1  declare stmt with decl label var
  v label var a DECLARE v LABEL; declare stmt with decl label var v.

18.1  non-declare stmt with ref labels-

18.2'  goto stmt with ref label f, label var f, assembler stmt

18.3'  if stmt with ref label f, label var f, fix-pt var v assembler label f assembler stmt

18.4'  a arith assign stmt with fix-pt var v assembler stmt

19.1'  stmt seq with stmt labels-

19.2'  stmt seq with stmt labels-

19.3'  declare stmt with decl label var

19.4'  declare stmt with decl label var

20.1'  stmt seq with stmt labels-

20.2'  legal stmt seq with stmt labels-

21.1  set of vars with assembler data stmt

21.2  PL/I program with translation

From the collection of the Computer History Museum (www.computerhistory.org)