*mentary formal systems*) are a variant of Post's canonical systems. In canonic systems the general framework of productions, or string-transformation rules, is replaced by a system of axioms (*canons*) and by the logical rules of substitution for variables and detachment (modus ponens). A canonic system defines a set of interrelated predicates, each of which is a set of strings.

Canonic systems have been used to specify the syntax and the translation of programming languages (More[3]; Donovan and Ledgard, 1967). They have served as a data base for a generalized translator for computer languages (Alsop, 1967; Altman[4]); later theorems as to their mathematical power and their formal properties have been proven (Doyle[4], Mandl[4]); and they have been used to study the complexity of translators and languages (Haggerty, 1969). The ultimate goal of this research has been to say something about programming languages. The author hopes to use this system to prove things about programming languages and their translators.

Consider the inadequacies of the Backus-Naur Form (BNF) specification of the syntax of programming languages. In BNF it is impossible to describe many of the constraints that exist in programming languages, such as the restriction that a "legal program" is not acceptable to a translator, even though correct in form, unless all of the reference labels in the program correspond to statement labels. These features are sometimes referred to as "context-sensitive features." Some people feel that they really refer to the meaning of a language and that they are semantic and not syntactic. However, these features *must* be specified in specifying the translation of a language. The distinction between syntax and semantics is not always clear. For example, the statement

```
20 GO TO 20
```

may be syntactically a legal statement, but is it semantically correct? (One might argue that this statement is useful for determining how long a computer will run before making an error.)

The translation of a programming language is specified by a canonic system that generates a set of ordered pairs of the form

*(statement in source language, its translation in the target language)*

Such a set of rules would specify the translation of a computer language. If the target language is understood, such a specification could be said to define the semantics of the language.

Like BNF, canonic systems *generate* the strings of a language. The *recognition* process is a different problem: An algorithm that uses a canonic system specification of a programming language to recognize and produce the transla-

[3] More, Yale classnotes, 1963.
[4] Unpublished M.S. Theses, M.I.T., 1970.

tion of strings has been developed and implemented.

The issue of whether or not canonic systems specifying the context-sensitive code parts of a programming language specify the syntax or the semantics of a language is academic; the real goal is to use a specification of a language to say something about the language and its translation. If we are to define the translation of a language, we must specify *all* the legal strings that get translated in that language. We need a system powerful enough to exclude illegal strings, whether on the basis of syntax or of semantics; hence, the motivation for a more powerful system than BNF. Yet, in their most general form, canonic systems are so powerful that they introduce many undecidability problems. A requirement exists, therefore, to determine more precisely the power of canonic systems and to restrict their power to the point at which they are adequate to handle the features we wish to define, but not powerful enough to introduce problems we cannot cope with in our models.

We will first introduce canonic systems informally. A canonic system consists of a number of *canons*, logical rules which state that certain *premises* imply certain *conclusions*. A *predicate* is a name given to a well-defined set of strings over the alphabet of the object language. For a programming language these sets are defined in such a way as to aid the user of the canonic system.

The assertion sign ⊢ is used to separate the premises from the conclusions. The general form of a canon is

a;b; . . . ;c ⊢z

and is read "from the premises a;b; . . . ; c can be asserted z." For example, we might define a set *number:*

⊢ 1 *digit*
⊢ 2 *digit*
⊢ 3 *digit*

*x digit* ⊢ *x number*
*x digit; y number* ⊢ *yx number*[5]

This system defines *number* as the set of strings over the symbols 1, 2, and 3. Any terminals may be substituted for the variables x and y, but no conclusion can be drawn unless the resulting premises are true – that is, unless the resulting premise statements have been previously reached as conclusions. The first three canons, which have no premises, are "axioms": their conclusions are immediate.

[5] When writing a canon the underline may be used in place of italics, e.g. x digit; y number ⊢ yx number

## PREDICATE, VARIABLE, TERM, REMARK, CANON

In the above example, *digit* and *number* are predicates of degree 1 that name certain sets of strings. A predicate of degree 2 names a set of ordered pairs of strings. A *term* is a string of concatenated variables and terminal symbols; a *remark* is a term followed by a predicate symbol. If $R_1, \ldots, R_{n-1} \vdash R_n$ is a canon, $R_1, \ldots, R_{n-1}$ are premises and $R_n$ is the conclusion. Each of the $R_i$ is a remark.

The following notation will be used:

1. Lower case letters will be used as variables.
2. Italicized strings of letters will be used as predicate symbols.
3. The notation $\langle x_1 < x_2 < \ldots x_n \rangle$ will be used to denote n-tuples. Terms of degree 1 will be denoted by their single component without the brackets.
4. A series of canons with identical premises $R_1, \ldots, R_n$ and different conclusions $\alpha_1 P, \ldots, \alpha_m P$ will be abbreviated (see Fig. 7.5 for example)

$$R_1; \ldots; R_n \vdash \alpha_{1+}\ \alpha_{2+}\ \alpha_{3+}\ \ldots\ \alpha_n P$$

## FORMAL DEFINITION

**Definition 12:** A canonic system is a sextuple

$$\mathscr{C} = (C, V, M, P, S, D)$$

where

- **C** is a finite set of *canons*
- **V** is an alphabet of terminal symbols used to form the strings generated (i.e., provable) by $\mathscr{C}$
- **M** is a finite set of *variable symbols* (variables)
- **P** is a finite set of *predicate symbols* (predicates) used to name sets of n-tuples. The number of components in the n-tuples denoted by a predicate is the *degree* of the predicate
- **S** is a finite set of punctuation signs used in writing canons
- **D** ($\subseteq$ P) is a set of *sentence predicates*, the union of which will be defined to be the *language specified by the canonic system*

The usual punctuation signs are $\vdash$, $\langle, \rangle$, $<$, and ;.

## SUBSTITUTION AND DETACHMENT

An *instance* of a canon is the result of substituting strings from V* for the variables that appear in the canon. Substitution must be uniform: occurrences of a single variable must all be replaced by the same string. The rule of inference (called *modus ponens*, or *detachment*) states that if $R_1; \ldots; R_{n-1} \vdash R_n$ is an

instance of a canon, the remark $R_n$ can be derived (in the canonic system) only if the premises $R_1; \ldots; R_{n-1}$ are all in the system. (Note that in the case of an "axiom" there are no premises). $R_n$ is then immediately derived from $R_1, \ldots, R_{n-1}$. A *proof* or derivation of a remark R in a canonic system $\mathscr{C}$ is a finite sequence of remarks $R_1, \ldots, R_n, R$ every member of which can be immediately derived from one or more of the preceding remarks. R is in $\mathscr{C}$ (can be derived or proven in $\mathscr{C}$) if and only if there exists a proof for R in $\mathscr{C}$.

### 7.6.1 Example: Syntax Specification

In this section we present an example of the use of canonic systems to specify the syntax of a programming language. Our example defines the same FORTRAN subset that was used to demonstrate the use of Backus Naur Form. A canonic system specification of this same subset is given in Figure 7.5.

1 $\vdash$ A+B+C . . . +Z *letter*
2 ℓ *letter* $\vdash$ ℓ *identifier*
3 ℓ *letter*; γ *identifier* $\vdash$ γℓ *identifier*
4 γ *identifier* $\vdash$ GO TO γ *go to stm*
5 i *identifier*; x *go to stm* $\vdash$ ix *program*
6 i *identifier*; x *go to stm*; p *program* $\vdash$ ixp *program*

**FIGURE 7.5** Canonic system specification of syntax

We can generate programs using this specification. For example, the string

A GO TO B

may be generated using the fifth canon with the terminal string substituted as below:

A *identifier*; GO TO B *go to stm* $\vdash$ A GO TO B *program*

A is in *letter* by the axiom (1) and so by (2) is in *identifier*. Therefore, the first premise of canon 5 is satisfied. The second premise may be satisfied by using canons 4, 2, and 1. Therefore, the following sequence using substitutions and modus ponens repeatedly derives A GO TO B *program*: (see Fig. 7.6).

| | | |
|---|---|---|
| 1 | A *letter* | C.1, MP |
| 2 | A *identifier* | 1,C.2, MP |
| 3 | B *letter* | |
| 4 | B *identifier* | |
| 5 | GO TO B *gotostm* | 4,C.4, MP |
| 6 | A GO TO B *program* | 5,2,C.5, MP |

(k, Cn, MP ~ Mp = Modus Ponens; Cn = canon in fig. 7.5; k = line in this fig.)

**FIGURE 7.6** Derivation of a string in a canonic system

We now construct a canonic system that specifies the same language but with the restriction that all reference labels appear among the statement labels. Thus the program below

```
A GO TO B
C GO TO D
```

will not be a legal program because the reference labels B and D are not among the statement labels A and C. To effect this restriction, we will eventually generate a set of ordered triples: a program, a list of reference labels in the program, and a list of statement labels in the program. Out of this set we want only the programs containing reference labels bearing the relationship *in* to the list of statement labels. To accomplish this, we rewrite the canonic system of Figure 7.5, as follows:

7 $\vdash A_+B_+C \ldots_+Z$ *letter*
8 $\ell$ *letter* $\vdash \ell$ *identifier*
9 $\ell$ *letter*; y *identifier* $\vdash$ y$\ell$ *identifier*
10 y *identifier* $\vdash$ ⟨GO TO y$_{<}$y⟩ *go to stm with ref label*

Canons 7-9 are similar to those used previously. However, 10 differs in that it defines a set of ordered pairs. Each member of this set consists of a pair of strings, GO TO y and y. That is, each element consists of a GO TO statement with a reference label, and the reference label. When generating the code for legal GO TO statements, we also must keep track of the reference labels.

11 s *identifier*; ⟨x$_{<}$r⟩ *go to stm with ref label* $\vdash$ ⟨s x$_{<}$s,$_{<}$r,⟩ *prog with stm labels and ref labels*

The above canon defines a set of ordered triplets. The first element of one of the members of this set is a string consisting of an identifier followed by a GO TO statement; the second element is the statement label; the third element is the reference label.

An instance of the canon scheme number 11 is

A *identifier*; ⟨GO TO MZ$_{<}$MZ⟩ *go to stm with ref label* $\vdash$
⟨A GO TO MZ$_{<}$A,$_{<}$MZ,⟩ *prog with stm labels and ref lables*

12 i *identifier*; ⟨x$_{<}\ell$⟩ *go to stm with ref label*;
⟨p$_{<}$s$_{<}$r⟩ *prog with stm labels and ref labels*
$\vdash$ ⟨i x p$_{<}$si,$_{<}$r$\ell$,⟩ *prog with stm labels and ref labels*

element is a program consisting of GO TO statements; the second element is a list of statement labels; and the third element is a list of reference labels (elements of the lists are separated by commas).

13 ⟨p<sub>s<sub>r⟩ *program with stm labels and ref labels;* ⟨r<sub>s⟩ *in* ⊢ p *program*

The above canon states that given an ordered triplet of which the first element is a program, the next element is a list of statement labels, and the third is a list of reference labels; given also that the list of reference labels bears the relationship *in* to the list of statement labels, then the program is a legal program. We now define the relationship *in* as a set of ordered pairs of which the first element is a list and the second element is a list of labels containing those of the first list.

Canons 14-16 define *list*:

14 ⊢ λ *list*
15 i *identifier* ⊢ i, *list*
16 x *list*; y *list* ⊢ xy *list*

Canons 17 and 18 define the predicate *in*:

17 x *list*; y *list*; z *list* ⊢ ⟨y<sub>xyz⟩ *in*
18 ⟨a<sub>ℓ⟩ *in*; ⟨b<sub>ℓ⟩ *in* ⊢ ⟨ab<sub>ℓ⟩ *in*

Canons 7-18 together define the set of legal programs.

### 7.6.2 Specification of Translation

Canonic systems may be used to specify the translation of a language. A translation is a function and may be defined by a set of ordered pairs, of which the first element is a legal program and the second element is its translation into the target language. For example, a specification of the translation of PL/I into 360 assembly language ultimately specifies the set of ordered pairs:

⟨legal PL/I program < 360 assembly-language program⟩

Figure 7.7 is a canonic system specifying the translation of the GO TO subset of FORTRAN into the assembly language of the IBM 360 (BAL). For simplicity of this specification, we have not included the restriction that reference labels be in the list of statement labels, nor, as would be the case in real BAL, have we limited the length of identifiers.

$\vdash A_+B_+C_+ \ldots Z$ *letter*
ℓ *letter* $\vdash$ ℓ *identifier*
ℓ *letter*; γ *identifier* $\vdash$ γℓ *identifier*
γ *identifier* $\vdash$ ⟨GO TO $\gamma_{<}$B γ⟩ *go to stm with translation*
ι *identifier*; ⟨$x_{<}y$⟩ *go to stm with translation* $\vdash$ ⟨$ix_{<}iy$⟩ *translation*
ι *identifier*; ⟨$x_{<}y$⟩ *go to stm with translation*; ⟨$p_{<}t$⟩ *translation* $\vdash$ ⟨$ixp_{<}iyt$⟩ *translation*

**FIGURE 7.7** A canonic system specification of the translation of a subset of FORTRAN to BAL

## Recognition and Translation Algorithm