Technical Report CS73002-R

# A COMPLETE HORIZONTAL MICROLANGUAGE

T. C. Wesselkamper*

and

Eric Nixon**

October 1973

*T.C. Wesselkamper, Dept. of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

**Eric Nixon, Dept. of Computer Science, University College, University of London, Gower Street, London WC 1

ABSTRACT

This paper defines a data space whose points are trees with leaves which are [name, value] pairs. Over this space a substitution operator S (meaning informally "in x for y put z") is formally defined. Taken together with several auxiliary operators, S is shown to be sufficient to define a large class of high level languages since S is known to be functionally complete for finite-valued spaces, a functionally complete language is exhibited.

# A Complete Horizontal Microlanguage

by

T. C. Wesselkamper
Virginia Polytechnic Institute and State University

and

Eric Nixon
University of London

Authors' addresses: T. C. Wesselkamper, Dept. of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061; Eric Nixon, Dept. of Computer Science, University College, University of London, Gower Street, London WC 1.

# 1. Introduction.

1.1 The notion of functional completeness has an extensive history and literature in pure mathematics. An analogous notion seems useful in discussing the relationship between a language and the primitives in terms of which it is defined.

Microcode does not appear to be a univalent term in computer literature. One man's code is another man's microcode. We use the terms language/microlanguage and code/microcode to describe entities which have the relationship definiendum/definiens. We describe a sequence of statements in a language as "code" and a sequence of statements in a microlanguage as "microcode." A microlanguage is complete for a language if each statement of the language can be defined as a sequence of statements of the microlanguage. A microlanguage is complete for a set of languages if it is complete for each language of the set. A microlanguage is horizontal if there exist statements in the microlanguage which contain more than one operator. It is clear that the relation of being complete for a language is a transitive relation.

Presumably the designers of any general purpose digital computer intend the instruction set of each machine to be complete for some rather large set of languages. This intention is realized by providing an instruction set so large that in some vague way one has the feeling that, armed with these instructions, anything is possible. While experience bears out this intuition, a rigorous proof of completeness might be difficult.

In this article we define a model of a data space and the syntax and semantics of a horizontal microlanguage over that space. We call this microlanguage Crampon. We show the microlanguage to be complete for a large class of languages.

Wesselkamper has studied an operator S defined by:

$$\underline{S}xyz = \begin{cases} z, \text{ if } x = y; \\ \\ x, \text{ if } x \neq y; \end{cases} \qquad (1.1.1)$$

and has shown that it is functionally complete over any finite valued space [4]. In this paper we generalize this definition to obtain a vehicle for defining high level languages, while adding this functional completeness, a property that no existing high level language has been shown to possess.

## 2. A Formalization of the Data Space and its Operators.

2.1 Let Q be the field of rationals under the operations # (addition) and · (multiplication). Let Q have the usual ordering under $<$. Let L be a denumerable set such that T, F, U $\in$ L and L $\cap$ Q = $\emptyset$. Let V = Q $\cup$ L. Let N be a denumerable set of names such that N $\cap$ V = $\emptyset$. The Cartesian product N x V is the set of [name, value] pairs with which we will frequently deal.

Suppose that in Q the additive inverse of an element a is denoted by $\bar{a}$ and the multiplicative inverse of $a \neq 0$ is denoted by $a^{-1}$. For x, y $\in$ V, define:

$$-x = \begin{cases} \bar{x}, \text{ if } x \in Q; \\ U, \text{ otherwise.} \end{cases}$$

$$x + y = \begin{cases} x \# y, \text{ if } x, y \in Q; \\ U, \text{ otherwise.} \end{cases}$$

$$x - y = \begin{cases} x \# \bar{y}, \text{ if } x, y \in Q \\ U, \text{ otherwise.} \end{cases}$$

$$x * y = \begin{cases} x \cdot y, \text{ if } x, y \in Q; \\ U, \text{ otherwise.} \end{cases}$$

$$x / y = \begin{cases} x \cdot y^{-1}, \text{ if } x, y \in Q, y \neq 0; \\ U, \text{ otherwise.} \end{cases}$$

$$\underline{\text{pos}}\ x = \begin{cases} T, \text{ if } x \in Q, \ 0 < x; \\ F, \text{ if } x \in Q, \ x < 0; \\ U, \text{ if } x \notin Q. \end{cases}$$

The elements T and F correspond to the logical values "true" and "false". The element U corresponds to the undefined or unassigned state.

2.2 A tree with root x is a finite graph such that each point except x is the end point of exactly one edge, x is the end point of no edge, and the graph contains no cycles. Such a graph is sometimes called an "arborescence." [1] Contrary to general usage we consider a single leaf to be a tree.

A tree A is a maximal subtree of a tree B if there exists no tree C such that A is a proper subtree of C and C is a proper subtree of B. If A is a maximal subtree of B we write $A \in B$.

The surtree of a tree A is the set of trees which result by deleting from the tree A its root. The surtree of A is the set of maximal subtrees of A. The surtree of A is an empty set only if A consists of a single leaf.

The notion of a tree is intuitively attractive, but it can be typographically unpleasant. For that reason, if $\{x_1, x_2, \ldots, x_n\}$ is the surtree of a tree A, we write:

$$A = (x_1, x_2, \ldots, x_n). \qquad (2.2.1)$$

A tree A with leaves in N x V possesses the unique name property if for each subtree B of A and each pair of leaves $[x, x'] \in B$, $[y, y'] \in B$, $x = y$ implies that $x' = y'$. (2.2.2)

It is clear from the definition that if a tree possesses the unique name property then each subtree of A possesses the unique name property also.

We define the data space X recursively as follows:

1. if $x \in N \times V$, then $x \in X$;
2. if $x_1, x_2, \ldots, x_n \in X$ and if $(x_1, x_2, \ldots, x_n)$ possesses the unique name property, then $(x_1, x_2, \ldots, x_n) \in X$. (2.2.3)

2.3 We define an operator $\underline{k}$ over $X \cup V$. On $N \times V$, $\underline{k}$ projects the $[\text{name, value}]$ pair onto its value coordinate. On V it is just the identity operator. Formally, if $n \in N$, $v \in V$, $x \in X$:

$$\underline{k}x = \begin{cases} v, \text{ if } y = [n, v]; \\ v, \text{ if } y = v; \\ U, \text{ if } x \notin N \times V. \end{cases} \qquad (2.3.1)$$

2.4 The effectiveness of the model which we are defining hinges upon the use of an operator which is adapted from the work of A.A. Markov.[2] It seems to have been first suggested by Thue.[3]

We denote this operator by $\underline{S}$. Each of its arguments is in $X \cup V$, as also is the result of the operation. In a naive way, $\underline{S}xyz$ is understood as "in x for the occurrence of y put z." Formally, for $x, y, z \in X \cup V$:

$$\underline{S}xyz = \begin{cases} z, \text{ if } x = y; \\ \left[x', z\right], \text{ if } x = \left[x', x''\right], y, z \in V, x'' = y; \\ (\underline{S}x_1yz, \underline{S}x_2yz, \ldots, \underline{S}x_nyz), \text{ if } y \in x, \\ \qquad x = (x_1, x_2, \ldots, x_n), \text{ and} \\ \qquad (\underline{S}x_1yz, \underline{S}x_2yz, \ldots, \underline{S}x_nyz) \in X; \\ x, \text{ otherwise} \end{cases} \tag{2.4.1}$$

This definition of $\underline{S}$ is a generalization of the definition noted in 1.1.1 and studied in 4. The definition of $\underline{S}$ on $X \times V$ is motivated, in part by the two following cases.

Firstly, if a, b, and c are all elements of $N \times V$, then the expression:

$$\underline{kSS}Tkabc \tag{2.4.2}$$

is equivalent to the McCarthy conditional[5] $(a \rightarrow b, c)$, for if $\underline{k}a = T$, $\underline{kSS}T\underline{k}abTc = \underline{kSS}TTbTc = \underline{kS}bTc = \underline{k}b$, whereas if $\underline{k}a \neq T$, $\underline{kSS}T\underline{k}abTc = \underline{kS}TTC = \underline{k}c$.

Secondly, if x and y are elements of N x V, the expression:

$$\underline{S}x\underline{k}x\underline{k}y \qquad (2.4.3)$$

is equivalent to the assignment statement x := y.

2.5 There are two operators which are analogous to set theoretic union and intersection. They perform the functions of creating new blocks and of adding new names to existing blocks, and of deleting names from blocks.

If $x, y \in X$ and $x \notin N \times V$, then for some elements $x_1, x_2, \ldots, x_n$ of X, $x = (x_1, x_2, \ldots, x_n)$. Define:

$$\underline{aug}\ x\ y = \begin{cases} (x_1, \ldots, x_n, y), \text{ if } (x_1, \ldots, x_n, y) \in X; \\ x, \text{ otherwise.} \end{cases} \qquad (2.5.1)$$

The expression $(x_1, \ldots, x_n, y)$ fails to be in X if it happens that y is a $\lceil$name, value$\rceil$ pair whose name already occurs as the name part of one of the $x_i$ in the surtree of x.

The deletion operator removes a leaf or a subtree from a tree. If $x \in X$ and $y \in N \cup X$, then define:

$$\underline{\text{del}}\ x\ y = \begin{cases} (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n), & \text{if } x = (x_1, \ldots, x_n) \\ & \text{and there exists a } y' \in V \text{ such that } \lceil y, y' \rceil = x_i \\ & \in x; \\ (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n), & \text{if } y = x_i \in x; \\ x, & \text{otherwise.} \end{cases} \qquad (2.5.2)$$

2.6 Suppose that a tree A has maximal subtrees B and C. Suppose further that $\lceil x, x' \rceil$, an element of N x V, is in the surtree of B and that there is no element $\lceil x, x'' \rceil$ in the surtree of C. Then B and C can be distinguished by the fact that $\lceil x, x' \rceil$ is in B. If, among the trees in the surtree of A, B is the only tree with a leaf containing the name x, then the name x <u>labels</u> B in A. Define:

$$\underline{\mathrm{bl}}_A x = \begin{cases} B, \text{ if there exists an } x' \in V \text{ such that } \left[x, x'\right] \\ \quad \in B \in A, \text{ and there exists no } C \in A \text{ and } x'' \in V \\ \quad \text{such that } \left[x, x''\right] \in C \in A; \\ (\left[x, U\right]), \text{ if there exists no } B \in A \text{ and } x' \in V \\ \quad \text{such that } \left[x, x'\right] \in B. \end{cases} \qquad (2.6.1)$$

In this paper there are only two contexts in which the operator $\underline{\mathrm{bl}}_A$ occurs. In each case the value of A can be inferred from the context. This enables us to write bl and to omit the A. The first of these contexts is as one of the three arguments of S. We define·

$$\underline{S}xy\underline{S}bl\underline{a}zw = \underline{S}xy\underline{S}\underline{\mathrm{bl}}_x azw$$

$$\underline{S}xbl\underline{a}z = \underline{S}x\underline{\mathrm{bl}}_x az$$

$$\underline{S}xyb\underline{l}a = \underline{S}xy\underline{\mathrm{bl}}_x a \qquad (2.6.2)$$

In words, if bl occurs as part of the first argument of S it refers to the first argument of the previous S; if bl occurs as part of the second or third argument of an S it refers to the first argument.

The second context in which bl can occur is as an augment of the aug operator. The aug operator, in turn, occurs only as a part of the third argument of an S. Define: –

$$\text{aug x bl y} = \text{aug x bl}_{x}\text{ y};$$

$$\text{Sxy aug bl zw} = \text{Sxy aug bl}_{x}\text{ zw}. \qquad (2.6.3)$$

Should there be a conflict between 2.6.2 and 2.6.3, the latter is the stronger convention.

2.7 The goto operator is a different kind of beast. All of the previous operators were defined as functions. The argument of goto is the evaluation of an S-string. Frequently the goto will take the form "goto kSSTkabTc." If $x \in V$, then "goto x" causes the statement labelled $[x]$ (the greatest integer in x) to be executed next. If $[x]$ does not exist or if no statement is labelled $[x]$, the execution is terminated.

We define a Crampon program to be a sequence of statements of the form:

$$\langle \text{label} \rangle \quad \langle \text{S-string} \rangle \quad ; \quad \langle \text{goto} \rangle \qquad (2.7.1)$$

where $\langle$label$\rangle$ is an integer followed by the character ":". We claim that a sequence of statements of this form defines each statement of a large class of high level languages.

The <S-string> of the Crampon expression will, in general, be a rather complex expression of the form Sxyz, where the arguments of the operator S are elements of X ∪ V. These arguments are themselves expressions involving S, k, bl, aug, del, pos and the four binary and one unary arithmetic operators. A formal BNF grammar for the syntax of Crampon forms an Appendix to this paper. (From that it may be noted that the infix arithmetic operators bind more strongly than the non-arithmetic prefix operators.) The constructions involved will be clear from the examples that follow.

We list in Table 1 the domains and ranges of each of the operators we have defined.

Table 1 -- Domains and Ranges of Crampon Functions

| Operator | Domain | Range |
|---|---|---|
| - (unary) | $V$ | $V$ |
| + | $V \times V$ | $V$ |
| - (binary) | $V \times V$ | $V$ |
| * | $V \times V$ | $V$ |
| / | $V \times V$ | $V$ |
| pos | $V$ | $\{T, F, U\}$ |
| bl | $X \times N$ | $X$ |
| aug | $X \times X$ | $X$ |
| del | $X \times (N \cup X)$ | $X$ |
| k | $X \cup V$ | $V$ |
| S | $(X \cup V)^3$ | $X \cup V$ |

## 3. The Completeness of Crampon for Algol-$\mathcal{E}$ .

3.1 In 1966 M. Nivat and N. Nolin published a subset of Algol which they designated as Algol-$\epsilon$ .[6] They showed that every Algol program could be converted into (defined in terms of) an Algol program. In our terms Algol-$\mathcal{E}$ is a complete microlanguage for Algol. We show that Crampon is a complete microlanguage for a language which is a slight modification of Algol-$\mathcal{E}$ .

Algol-$\epsilon$ is described as follows:

> The structure of programs written in this language is the same as in ALGOL 60: declarations, statements, parentheses, begin, and end play their same parts.
>
> The declarations are those of ALGOL 60 with a few restrictions, the most important of which concerns the array declarations; the limits of array declarations can only be expressed by constants or simple variables.
>
> In order to define the statements authorized by ALGOL-$\mathcal{E}$ conveniently, we shall call a "term" a simple variable or an indexed variable whose indices are constants or simple variables.

With this in mind, we state the following definitions: -

1) Assignment statements are of the form

$$a := B$$

where a is a term and B is either a nonconditional expression (with at most one connector and, therefore, at most two terms) or an expression of the form

$$\gamma ( \delta_1, \delta_2, \ldots, \delta_n )$$

where $\gamma$ is a procedure identifier and $\delta_1, \ldots, \delta_n$ are terms.

2) Unconditional jump statements are of the form

goto a

where a is a term (with at most one index).

3) Conditional jump statements have the forms

if a then $T_1$

or

if a then $T_1$ else $T_2$

where a is a term and $T_1$, $T_2$ are unconditional jump statements.

4) Procedure statements have the form

$$\Upsilon(\delta_1, \delta_2, \ldots, \delta_n)$$

where $\Upsilon$ is a procedure identifier and $\delta_1, \ldots, \delta_n$ are terms.

There are no other elementary statements in ALGOL-$\varepsilon$ . Blocks and procedure bodies are defined as in ALGOL 60; compound statements are no longer of any interest; all statements may be labelled. [6, pp. 148-9.]

3.2 We modify this description in two significant ways. One modification is a restriction, the other is a concession.

The "Revised Report on the Algorithmic Language ALGOL 60." states: "Identifiers (except those representing labels) occurring within a block and not being declared to this block will be non-local to it, i.e., will represent the same entity inside the block and in the level immediately outside it." [7, paragraph 4.1.3.]

We require that each variable used in a block be declared to the block. As a result an variable which is to be non-local must be passed as a parameter to a procedure.

It is possible to build into Crampon a mechanism which corresponds to the Algol property of non-local names. We have done this elsewhere [8]. We eliminate it here in the interest of brevity.

As a compensation for the above loss of non-local variables, we allow a declaration to appear at any point in a program.

As a result of the above limitation on the scope of names, we may employ the abbreviation of using a name to refer to the corresponding $[\text{name, value}]$ pair. Thus $[y, y']$ may be abbreviated y. Note that since, by 2.3.1, $\underline{k}\,[y, y'] = y'$, our abbreviation yields $\underline{k}y = \underline{k}\,[y, y'] = y'$. Thus the may say that y is an abbreviation for $[y, \underline{k}y]$.

3.3 In Section 2.7 we defined a Crampon statement to be of the form:

$$\langle \text{label} \rangle \langle \text{S-string} \rangle \quad ; \quad \langle \text{goto} \rangle$$

We show now how an Algol-$\epsilon$ statement is defined in terms of such a Crampon expression.

We adopt the convention that every block of an Algol-$\epsilon$ program is labelled. Suppose that the outermost block of the program bears the label L1. Suppose that an Algol statement is contained in some inner block as indicated by the fragment:

```
L1: begin
      .
      .
      .
    L2: begin
          .
          .
          .
        Ln: begin
              .
              .
              .
              <statement>
              .
              .
              .
            end
          .
          .
          .
        end
end
```

We suppose that the environment in which the program exists creates the point ( [ L1, U ]) when it encounters the program. We refer to the current value of the space as bl L1, since bl L1 is not otherwise defined by 2.6.1 and 2.6.2.

The prefix

S bl L1 bl L2 S bl L2 bl L3 S ... S bl Ln-1 bl Ln S bl Ln ... (3.3.1)

precedes any Crampon expression which defines the specific

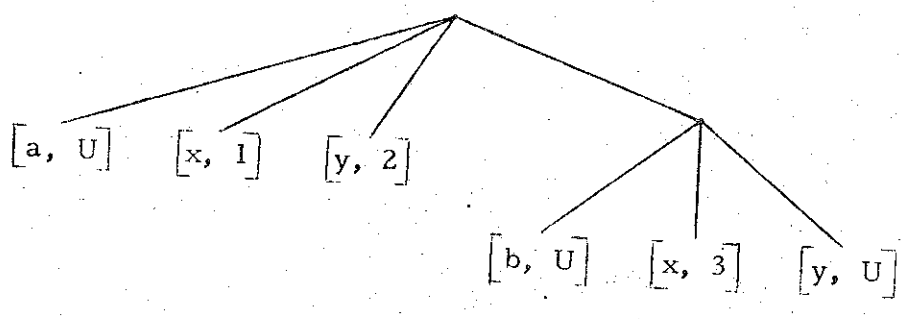
statement. This prefix effectively locates the particular block of the program in which the statement occurs.

Specifically, consider the Algol fragment:

```
a: begin real x, y;
   x  := 1;

   y  := 2;

b: begin real x, y;

   x  := 3;

   y  := 4;
```

(3.3.2)

Immediately before the execution of the statement 'y := 4;' the state defined by the fragment is modelled by the data point:



The statement 'y : = 4;' is equivalent to the Crampon expression:

$$\underline{S}\ \underline{bl}\ a\ \underline{bl}\ b\ \underline{S}\ \underline{bl}\ b\ y\ \underline{Sy}\ \underline{ky}\ 4\ . \qquad (3.3.3)$$

Here we have:

$$\underline{bl}\ a = (\lceil a, U\rceil, \lceil x, 1\rceil, \lceil y, 2\rceil, (\lceil b, U\rceil, \lceil x, 3\rceil, \lceil y, U\rceil));$$

$$\underline{bl}\ b = (\lceil b, U\rceil, \lceil x, 3\rceil, \lceil y, U\rceil);$$

$$y = \lceil y, U\rceil;$$

$$\underline{ky} = U\ .$$

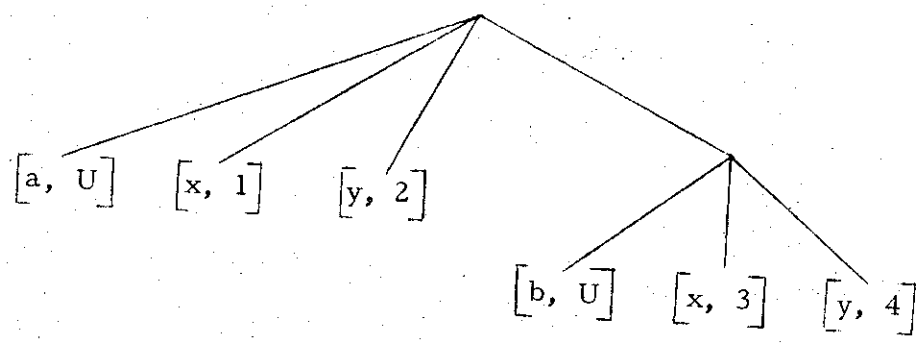
Successive applications of the definition of $\underline{S}$ is 2.4.1 yield:

$$\underline{Sy}\ \underline{ky}\ 4 = \underline{S}\lceil y, U\rceil\ U\ 4 = \lceil y, 4\rceil;$$

$$\underline{S}\ \underline{bl}\ b\ y\ \lceil y, 4\rceil = S(\lceil b, U\rceil, \lceil x, 3\rceil, \lceil y, U\rceil)\lceil y, U\rceil\ \lceil y, 4\rceil$$
$$= (\lceil b, U\rceil, \lceil x, 3\rceil, \lceil y, 4\rceil);$$

$$\underline{S}\ \underline{bl}\ a\ \underline{bl}\ b\ (\lceil b, U\rceil, \lceil x, 3\rceil, \lceil y, 4\rceil)$$
$$= (\lceil a, U\rceil, \lceil x, 1\rceil, \lceil y, 2\rceil, (\lceil b, U\rceil, \lceil x, 3\rceil, \lceil y, 4\rceil))$$

or more graphically:




This point models the state defined by the Algol fragment at its completion.

The general cases of the assignment statement, for example, x : = y * z; x : = -y; x : = y;, behave in exactly the same fashion. As noted in Section 2.7, the infix arithmetic operators bind more strongly than the non-arithmetic prefix operators. As a result the Crampon expression 'Sx kx ky * kz' is unambiguous.

The last expression of each Crampon statement is a < goto > . The Crampon equivalent to the Algol-$\varepsilon$ statement 'if a then go to b else go to c;' is given by:

; kSST ka b T c . (3.3.4)

In the formal definition of Crampon the < goto > expression may be explicit in each statement or may be empty. In the latter case

it is understood to mean that the next statement sequentially is executed next.

An operator equivalent to the Algol relational "=" (the Fortran .EQ.) does not have to be introduced separately. The Algol relation "x = y" is given by the expression:

SSSTSTxFFTSTSTyFFFSSxyTxF. (3.3.5)

An Algol label "behaves as though declared at the head of the smallest embracing block [7, paragraph 4.1.3]." Entry into a block results in the augmentation of that block by a point which represents each label in the block as well as by a point representing each name declared in the block head. In the case of a label a, the block is augmented by the point ([a, U]); in that case of a name x, the block is augmented by the point [x, U]. Consider again the fragment of 3.3.2. By definition, the environment in which the computation exists contains the point ([a, U]) and this we refer to as bl a. The declaration "real x, y;" together with the use of the label b in the block defines an effect equivalent to the Crampon expression:

S bl a bl a aug aug aug [x, U] [y, U] ([b, U]) (3.3.6)

which transforms the initial data point ([a, U]) into the point ([a, U], [x, U], [y, U], ([b, U])).

The corresponding exit from this block causes the deletion of the [name, value] pairs corresponding to the names declared in the block, as well as the deletion of labels used in the block. In this case the deletion is effected by:

$$\underline{S}\ \underline{bl}\ a\ \underline{bl}\ a\ \underline{del}\ \underline{del}\ \underline{del}\ x\ y\ \ ([b,\ U]) \qquad (3.3.7)$$

Should the block entered or left occur at a deeper level in the block structure the analogous Crampon definition begins with a prefix of the form of 3.3.1.

As would be expected in a low level language, the effect of procedures usage is achieved by copying the text of the procedure (as is done with a macro). In this the Algol call by name produces no difficulty. The effect of a call by value is "as though an additional block embracing the procedure body were created in which" the formal parameters of the procedure declaration "are assigned the values of the corresponding actual parameters [7, paragraph 4.7.3.1]." This effect can be created in a Crampon program by declaring, at the time of procedure use, new names for the formal parameters (with suitable substitution if the names of the formal parameters duplicate names which have been previously declared in the block from which the procedure is called. Upon exit from the procedure the names and values corresponding to the formal parameters are deleted.

Thus we have completed the informal sketch of the proof that Crampon is complete for Algol-$\mathcal{E}$. By transitivity, Crampon is complete for Algol.

## 4. Conclusions.

The interest of Crampon arises not from the fact that it is yet another vehicle for defining or implementing an Algol compiler, but from the fact that it is based upon a generalization of the S operator which, as noted in Section 2 and proved in [4], is functionally complete for any finite space. Theoretically this is a great advantage.

The practical value of the approach is currently being investigated via a microprogrammed emulation on an H-P 2100 A computer.

## Appendix -- A BNF Grammar for Crampon

<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

<identifier> ::= <letter> | <identifier> <letter>

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

<number> ::= <digit> | <number> <digit>

<logical> ::= T | F | U

<op> ::= + | - | * | /

<array name> ::= <identifier> &

<array element name> ::= <array name> <index>

<index> ::= [<operand>]

<name> ::= <identifier> | <array element name>

<value> ::= <real> | <logical>

<real> ::= <number> . <number>

<operand> ::= <value> | <token>

<operation> ::= [<operand> <op> <operand>]

<token> ::= k <pair> | k <name>

<pair> ::= [<name> , <value>]

<argl> ::= <arg>

<arg2> ::= <arg>

<arg3> ::= <arg> | <block change> | <operation>

<arg> ::= <S-string> | <block> | <pair> | <name> |
<operand> | <positive>

<positive> ::= pos <operand>

<block> ::= (<element list>) | bl <name>

<element list> ::= <element list> , <element> | <element>

<element> ::= <pair> | <block>

<block change> ::= <augmentation> | <deletion>

<augmentation> ::= aug <augmentation> <aug tail> |
aug <A-block> <aug tail>

< aug tail > : := < pair > | < block >

< deletion > : := del < deletion > < del tail > |

del < D-block > < del tail >

< A-block > : := < block >

< D-block > : := < block >

< del tail > : := < pair > | < block > | < name >

< S-string > : := S < arg1 > < arg2 > < arg3 > | < empty >

< label > : := < number > :

< goto > : := < number > | < token > | k < S-string > | < empty >

< statement > : := < label > < S-string > ; < goto >

< statement list > : := < statement > | < statement list > < statement >

< program > : := < statement list >

References


1.) C. Berge, The Theory of Graphs and its Applications, (New York: Wiley, 1962), p. 160.

2.) A.A. Markov, Theory of Mathematical Algorithms, (Jerusalem: Israel Program for Scientific Translations, 1962), pp. 192-222.

3.) Axel Thue, "Probleme uber Veranderungen von Zeichenreihen nach gegebenen Regeln", Skrifter utgit av Videnskapsselskapet i Kristiana, I. Matematisk-naturvidenskalbelig klasse 1914, no. 10 (1914), 34 pp.

4.) T.C. Wesselkamper, "A Sole Sufficient Operator", Notre Dame Journal of Formal Logic, (to appear).

5.) John McCarthy, "A Basis for the Mathematical Theory of Computation", Computer Programming and Formal Systems, edited by P. Braffort and D. Hirschberg, (Amsterdam: North Holland Publishing Company, 1963), pp. 33-70.

6.) M. Nivat and N. Nolin, "Contribution to the Definition of Algol Semantics", Formal Language Description Languages for Computer Programming, edited by T.B. Steele, Jr., (Amsterdam: North Holland Publishing Company, 1966), pp. 143-57.

7.) Peter Naur (editor), "Revised Report on the Algorithmic Language ALGOL 60", The Computer Journal 5 (1962-1963) pp. 349-67.

8.) T.C. Wesselkamper, A Mathematical Model of the Computing Process in a High Level Language (unpublished Ph.D. thesis), (London: University of London, 1972), pp. 41-43.