STANFORD ARTIFICIAL INTELLIGENCE PROJECT
Memo No. 19

AUGUST 14, 1964

PROGRAMMING LANGUAGES AND TRANSLATION

by Jan Hext

Abstract: A notation is suggested for defining the syntax of a language in abstract form, specifying only its semantic constituents. A simple language is presented in this form and its semantic definition given in terms of these constituents. Methods are then developed for translating this language, first into a LISP format and from there to machine code, and for proving that the translation is correct.

The research reported here was supported in part by the Advanced Research Project Agency of the Office of the Secretary of Defense (SD-183)

PROGRAMMING LANGUAGES AND TRANSLATION

by Jan Hext

1. Languages and Translation:

As is widely appreciated, ALGOL has an adequate formal syntax, but its semantic description leaves much to be desired. Moreover, until such semantics are established on a formal basis, it will not be possible to develop any theory of computation in the field of languages and compilers. This paper explores one or two lines along which such a theory might be established. It is a development of the ideas in sections 12 and 13 of [1].

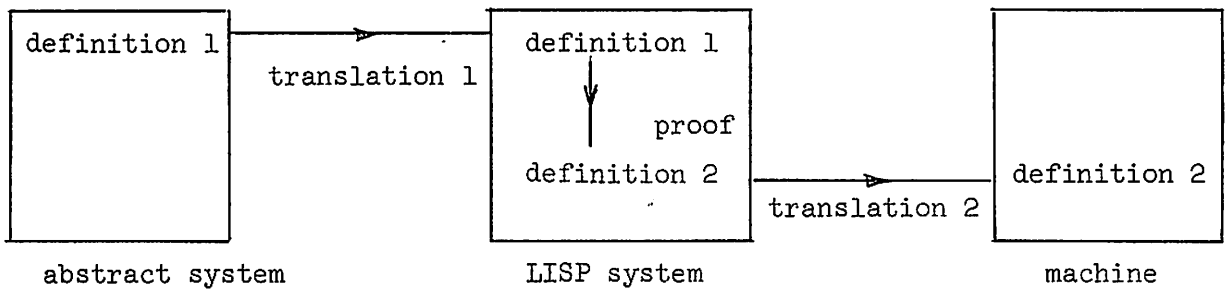
A language will be defined in terms of a system. This may be an abstract system, or a LISP system, or a machine system, or any system we care to define. Its constituents are names, numbers and a set of operators (or, if we care to make the distinction, representations of these objects). It also contains a set of predicates which classify subsets of these constituents: e.g., plusterm [a+b] might be true. For each class there are analysis functions which yield the essential elements of a member: e.g. first and second might yield a and b in the above example.

A language will be defined within the system as a function of the predicates and analysis functions.

An equivalent system will be one which has a representation of the same basic objects and the same structure (i.e., with one-one correspondence) of predicates and analysis functions. The language may then be given an identical definition within this system, simply by replacing the predicates and analysis functions of the first system by their equivalents in the second. It is easy to specify and prove a correct translation from one system to an equivalent system when the definition is thus preserved.

The definition of an object code program, however, will have a different form. So at some stage we must move to this form of definition and prove that it gives the same results. This is done, in the example below, within the LISP system. It is not part of the translation process; but it is the central part of proving that the translation is accurate.

The approach may be illustrated by the following diagram. In practice, things do not turn out quite so simply, but the underlying scheme is the same.



In sections 2 - 7, the abstract system, the LISP system, definition 1 and translation 1 are given in detail for a very simple language. Sections 8 - 9 give some aspects of the proof and of translation 2. Section 10 looks at some possible criticisms.

2. Syntax Format

The syntax of the language is given by statements of the form

$$\begin{aligned} \text{< class >} ::= \ & \text{pred 1} \rightarrow \{\text{fn 1a < class 1a >}, \ldots, \text{fn 1x < class 1x >}\}, \\ & \text{pred 2} \rightarrow \{\text{fn 2a < class 2a >}, \ldots, \text{fn 2y < class 2y >}\}, \\ & \vdots \\ & \text{pred n} \rightarrow \{\text{fn na < class na >}, \ldots, \text{fn nz < class nz >}\} \end{aligned}$$

pred 1, ... pred n are predicates of the system, defined for all items in it; fn 1a, ... fn nz are analysis functions of the system, defined for all items which satisfy their corresponding predicate.

The analytic meaning of the statement is that, if M is a member of the class, then it satisfies one of the predicates; and that the associated analysis functions, operating on M, will then yield components of the corresponding classes.

The synthetic meaning of the statement is that M is a member of the class if it satisfied one of the predicates and if it then has components of the corresponding classes. We may write this as

$$\begin{aligned} \text{valid [M, class]} = \ & (\text{pred 1 [M]} \wedge \text{valid [fn 1a [M], class 1a]} \wedge \ldots) \\ & \vee (\text{pred 2 [M]} \wedge \ldots) \\ & \vdots \\ & \vee (\text{pred n [M]} \wedge \ldots \wedge \text{valid [fn nz [M], class nz]}) \end{aligned}$$

We assume that valid [N, number] and valid [N, name] are defined for all N within the system, since < number > and < name > are basic (or terminating) classes. If any other terminating classes are included, then valid is assumed to be defined on them also.

3. Example : Micro-algol

```
< program > ::= yes → { declarations < declarations >,
                        commands < commands > }
```

A program has only one structure, satisfying the predicate yes. It yields a set of declarations and a set of commands. We have used the class names as the names of the analysis functions; it is often convenient to do this, and in such cases we conventionally omit the class name. The above, for example, is abbreviated to

```
< program > ::= yes → { declarations, commands }
```

The rest of the syntax is self-explanatory.

```
< declarations > ::= none → { }, some → { first < name >,
                                        rest < declarations > }
< commands > ::= none → { } , some → { labels, first < command >,
                                        rest < commands > }
< labels > ::= none → { } , some → {first < name >, rest < labels > }
< command > ::= stop → { } ,
                jump → { expression,name } ,
                assign → { name, expression }
< expression > ::= constant → { number } ,
                   variable → { name } ,
                   sum → { first < expression >, second < expression > } ,
                   product → { first < expression >, second < expression > }
```

< name > and < number > are basic classes.

4. Definition of Micro-algol

The definition of a language is a series of functions which define the results of all valid programs. A non-convergent function corresponds to a non-terminating program. Before giving the criteria for a valid definition, it is helpful to consider an example, in order to discover what the basic concepts are. We give the semantic definition of micro-algol, corresponding to the above syntax.

```
algol [ < program > P ] = al [B, init [declarations [P]], B ]

                          where B = commands [P]
```

```
al [ < commands > B, < table > T, < commands > C ] =
        none [C] → error,
      ( stop [f] → T,
        jump [f] → al [B, T, (value [expression [f],T] > 0 →
                                  dest [name [f],B], rest [C] ) ],
        assign [f] → al [B, store [name [f], value [expression [f],T],
                                   T], rest [C]  ]
                                   where f = first [C]  )
dest [ < name > N, < commands > C ] =
        none [C] ∨ among [N, labels [labels [C]] → C, dest [N, rest [C]]
among [ < name > N, < labels > L ] =
        some [L] ∧ (N = first [L] ∨ among [N, rest [L]])
value [ < expression > E, < table > T ] =
        constant [E] → number [E],
        variable [E] → fetch [name [E], T],
        sum [E] → value [first [E],T] + value [second [E], T],
        product [E] → value [first [E],T] x value [second [E],T]
```

In this definition, we require that the functions init, fetch, store be subject to the following conditions:

```
belong [ < name > N, init [< declarations > D]] = among [N,D]
    where among [N,D] = some [D] ∧ (N = first [D] ∨ among [N, rest [D]] )
fetch [ < name > N, init [< declarations > D]] = among [N,D] → empty, error
belong [< name > M, store [< name > N, < number > V, < table > T]] =
                                   belong [M, T]
fetch [< name > M, store [< name > N, < number > V, < table > T ]] =
       belong [M,T] → ( N=M → V, fetch [M,T] ), error
```

Here empty and error are elements of the class < number > ; empty is the initial value of a declared variable ; error is the permanent value of an undeclared variable.

5. Definition Completeness

By completeness we mean that algol [program] is well-defined for all valid programs. In other words, at no stage of evaluation is an argument outside the range of its function. We develop criteria for completeness along the following lines.

(1) The following operators are assumed to be defined over their conventional ranges: + x → , ∨ ∧ = Equality is defined on names as well as numbers.

(2) An expression is taken to be a function (or operator) applied to its arguments; the arguments may themselves be expressions. It is defined if the arguments are all within the range of their functions. We state the following:

(a) λ(< range > args) E (args)

is defined over the given range if the expression E is defined for all args within that range.

(b) label (f) (λ(< range > args) E (f, args))

is defined over the given range if it is primitive recursive and if E is defined for all args within that range. In checking that E is defined, we assume, of course, that f is defined.

(3) The predicates and analysis functions are defined over the ranges stated above.

(4) The definition of micro-algol includes items of the class < table >. This class is not defined in the syntax, but it is helpful to define it in syntactic form. Basically it has only one type of structure, and this yields two components, a belong function and a fetch function. We therefore define it b the statement < table > ::= yes → { belong < fn: name → truth > , fetch < fn: name → number > }. Thus belong [table] [name] gives a value either true or false; it will be conventional to write it as belong [name, table]. Similarly for fetch. This is the notation used in the definition of micro-algol above.

Associated with this type are two functions init and store. These have results of class table, and so it is sufficient to define them by specifying their resultant belong and fetch functions. This is done by the last four statements of the definition of algol.

Using (1) - (4), it can now be verified that algol is complete, in the sense given. The axioms and induction principles must be formulated before a formal proof can be given.

6. An Equivalent System in LISP

We proceed to set up an equivalent system in LISP. This is possible because LISP provides the necessary components - names, numbers and a set of items, namely S-expressions. The equivalence of names and of numbers within the two systems is assumed. The lines along which we proceed, in setting up our criteria of equivalence, follow those of (1) - (4) above. We use " $\sim$ " to denote equivalence;

the prime, " ' " , will denote items in the LISP system.

(1) Each of the following operators is assumed to be equivalent in the two systems: + x → ,∨∧ =

(2) Two expressions are equivalent in the two systems if they consist of equivalent functions applied to equivalent arguments. We state the following criteria for equivalent functions:

(a) λ(args) E (args) ~ λ(args') E' (args')
if args ~ args' => E ~E'.

(b) label (f) (λ(args) E (f, args)) ~ label (f') (λ(args') E'(f', args')) if the definitions are primitive recursive and if f~ f' ∧ args~args' => E ~ E'.

(3) We define the predicates and analysis functions in the LISP system in terms of LISP operators - null, car, cdr, etc. We set up a one-one equivalence correspondence with those of the abstract system. We then say that two items are equivalent if they satisfy equivalent predicates and have equivalent components. This corresponds closely with the definition of valid; it defines equivalent classes in the LISP system.

The following are the equivalents. The predicates one, two, three in the LISP system are true for lists with one, two, three members; some is true for any non-empty list.

Class	Abstract	LISP
< program >	yes, declarations, commands	two, car, cadr
< declarations >	none	null
	some, first, rest	some, car, cdr
< labels >	none	null
	some, name, rest	some, car, cdr
< commands >	none	null
	some, labels, first, rest	three, car, cadr, caddr
< command >	stop	one ∧ car = STOP
	jump, expression, name	three ∧ car = JUMP, cadr, caddr
	assign, name, expression	three ∧ car = ASSIGN, cadr, caddr
< expression >	constant, number	two ∧ car = NUMBER, cadr
	variable, name	two ∧ car = NAME, cadr
	sum, first, second	three ∧ car = PLUS, cadr, caddr
	product, first, second	three ∧ car = TIMES, cadr, caddr

(4) We could now define equivalents of init, store, belong, fetch as explicit LISP functions and show that the resulting functions, belong [init], etc., are equivalent under (2).

However, this is to some extent a confusion of the system notions of equivalence and the definition notions of equivalence. We therefore adopt the following approach: it effectively puts the abstract system on a more formal footing, and suggests a way for defining equivalence of definitions.

(i) Add to the syntax the following class definitions:

< table > ::= none → { }, some → { first < entry > , rest < table > }

< entry > ::= yes → { name, number }

(ii) Define

belong [< name > N, < table > T] = null [T] → false,
some [T] → (name [first [T]] = N ∨ belong [N,rest[T]])

fetch [< name > N, < table > T] = null [T] → error,
some [T] → (name [first[T]] = N → number [first[T]],
fetch [N, rest [T]])

(iii) Define init and store in terms of abstract construction statements:

init [< declarations > D] = Construct < table > : none[D] → none [init],
some[D] → some[init] ∧ first[init] = set[first[D], empty] ∧ rest[init] = init [rest [D]]

set [< name > N, < number > V] = Construct < entry > :
yes [set] ∧ name [set] = N ∧ number [set] = V

store [< name > N, < number > V, < table > T] = Construct < table > :
none [T] → none [store],
some [T] → some [store] ∧ (name [first [T]] = N →
first [store] = set [N,V] ∧ rest [store] = rest [T] ,
first [store] = first [T] ∧ rest [store] = store [N,V,rest[T]])

The meaning of these statements is self-evident: the RHS states that the result of the construction must satisfy a condition dependent on its arguments. We must establish that store always converges.

Thus we replace the four conditions at the end of the definition of algol by the above definitions, and claim equivalence. By this we here mean that the new definitions satisfy the old conditions. For

example, we must verify that under the above definitions

$$\text{fetch}[M,\text{store}[N,V,T]] = \text{belong}[M,T] \rightarrow (N = M \rightarrow V,\ \text{fetch}[M,T]),\ \text{error}$$

We develop the first part of the proof; "store" is short for "store [N,V,T]".

Suppose 1: belong [M,T] $\wedge$ N = M

=> some [T] $\wedge$ (name[first[T]] = M $\vee$ belong [M,rest [T]])

=> some [T] $\wedge$ (name[first[T]] = N $\vee$ belong [N,rest [T]])

Suppose 1.1: some [T] $\wedge$ name [first[T]] = N

=> some [store] $\wedge$ first[store] = set [N,V] $\wedge$ rest[store] = rest [T]

=> fetch [M,store] = N = M $\rightarrow$ V, fetch [M,rest[T]]

=> fetch [M,store] = V

Suppose 1.2: some [T] $\wedge$ name[first [T]] $\neq$ N $\wedge$ belong [N,rest [T]]

=> some [store] $\wedge$ first [store] = first [T] $\wedge$ rest[store] = store[N,V,rest [T]]

=> name [first [store]] $\neq$ N

=> name [first [store]] $\neq$ M

=> fetch [M,store] = fetch[M,store[N,V,rest[T]]

=> fetch [M,store] = V - by recursion induction.

This last deduction is made because we have proved a result of the form

$$\text{fetch}[T] = \text{pred}[T] \rightarrow (\text{cond}[T] \rightarrow\ ,\ \text{fetch}[\text{rest}[T]])$$

where cond[T] => rest [T] exists $\wedge$ pred[rest[T]]

The justification and completion of this proof is left as an exercise to proof-checkers!

(5) We have thus introduced a new feature into the abstract system, namely the construction feature. A construct function will require a corresponding function in the LISP system, which we define to be equivalent if it satisfied equivalent conditions.

As an example, we define init in the LISP system and prove its equivalence. We have in the abstract system

init [< declarations > D] = Construct < table >: none[D] → none[init] ,

some[D] → some[init] ∧ first[init] = set[first[D], empty] ∧ rest [init] = init[rest[D]]

We set up some further equivalent predicates and analysis functions for the classes < table > and < entry >.

Class	Abstract	LISP
< table >	none	null
	some, first, rest	some, car, cdr
< entry >	yes, name, number	two, car, cadr

We now define in the LISP system

init [< declarations > D] = null[D] → NIL,
some [D] → cons[set[car[D], empty], init[cdr[D]]]

set [< name > N, < number > V] = list[N,V]

The equivalent of the above construct condition is therefore

null[D] → null[init], some [D] → some[init] ∧ car[init] = list[car[D], empty] ∧cdr[init] = init[cdr[D]]

and this is clearly satisfied.

The definition of store is

store [< name > N, < number > V, < table > T] = null[T] → T, some [T] → (caar[T] = N → cons[list[N,V], cdr[T]], cons[car[T], store[N, V, cdr[T]]])

7. Translation to the LISP System

We have now set up a LISP system equivalent to the abstract system. The definition of algol within the LISP system is simply the functions algol, al, dest, among, value with their basic functions replaced by their LISP equivalents.

We now specify the translation of a valid abstract program into a valid LISP program by a function rep. This is a valid translation if

program ∼ rep[program]

because the function algol in the LISP system will then produce equivalent results.

```
rep [ < program > P ] = yes[P] → list[declarations[P]], rep[commands[P]]]
rep [ < declarations > D ] = none [D] → NIL,
                             some[D] → cons[first[D], rep[rest[D]]]
rep [ <commands> C] = none[C] → NIL,
                      some[C] → list[rep[labels[C]],rep[first[C]],
                                     rep[rest[C]]]
rep[ < labels > L] = none [L] → NIL,
                     some [L] → cons[name[L], rep[rest[L]]]
rep[ < command > S] = stop[S] → list[STOP],
                      jump[S] → list[JUMP,rep[expression[S]], name[S]],
                    assign[S] → list[ASSIGN, name[S], rep[expression[S]]]
rep[ < expression > E] = constant [E] → list [NUMBER E] ,
                        variable[E] → list[NAME, E],
                        sum[E] → list[PLUS, rep[first[E]], rep[second[E]]],
                     product[E] → list[TIMES, rep[first[E]], rep[second[E]]]
```

To show that P ~ rep[P], we show first that E ~ rep[E] using recursion induction; and work backwards through S,L,C,D. Each stage is a simple application of the above criteria.

8. An Equivalent Definition

The need at this stage is to re-define algol in terms of much smaller steps, corresponding more closely to actual machine orders. In fact, we reduce the steps to a set of macros, each corresponding to a short sequence of machine orders. At the same time, we introduce the concept of a stack, a list of program constants and parameters for the labels. The final correspondence will be with assembly code, which will include a parameter facility for labelling orders. No attempt is made at optimization, though this could certainly be included if required.

We begin by making a list, C, of constants within the program. Every constant in the program is then replaced by its position in this list (a relative address). Likewise, every occurrence of a variable in the program is replaced by a reference to its position in the list of variables, i.e., the table T.

Similarly, we form a list of labels, L, and replace their occurrences by references to their positions in the list (their parameters).

We define the following macro operators:

LOADV - load a variable on to the top of the stack.
LOADC - load a constant on to the top of the stack.
JUMP - fetch top of stack and jump if positive.
ADD - replace top 2 items on the stack by their sum.
MULT - replace top 2 items on the stack by their product.
STORE - fetch top of stack and store in the table.
STOP - stop.

A complete macro is an S-expression of the form (operator.argument), corresponding to an order and an address.

We would therefore prefer at this stage that the basic forms of the program l included the following:

```
assignment : (expression (STORE.name))
transfer : ( expression (JUMP.name))
expression : (LOADC.number)
             (LOADV.name)
             (expression   expression   (ADD.NIL))
             (expression   expression   (MULT.NIL))
```

This is achieved by a simple re-writing of the previous definitions of rep and the predicates and analysis functions. The names and numbers are then replaced by their positions in T, C and L.

We now define algol so that it steps through the macros in the obvious order, taking the specified action. It is necessary to keep track of all the branch points down which we move: A will denote the current position, and B the list of branch points. If we then move down car[A], we must add cdr[A] to B. The definition therefore takes the following form:

algol [< program > P] = al [P, definitions [P], constants[P], labels[P], $\emptyset$, commands [P], NIL]
where al [P, T, C, L, S, A, B] = ...

Here $\emptyset$ denotes an empty stack vector.

We establish the equivalence of the definition by showing that the execution of a command leaves C, L, S, and B unaltered (S will be back to $\emptyset$ and B back to NIL) while the table T and the position A are changed exactly as in the original definition. The proof is built up as a series of theorems. We state and prove the first:

Theorem If car [A] = rep [< expression > E] then
al [P,T,C,L,S,A,B] = al [P,T,C,L, load [value[E],S], cdr [A],B]

Proof We prove the case when E is an addition expression, of the form

(second first (ADD.NIL))

Using implied definitions of al we will have

al[P,T,C,L,S,A,B] = al[P,T,C,L,S, car [A], cons[cdr[A],B]]
since car [A] is a list

= al[P,T,C,L, load[value[second],S], cdar[A], cons[cdr[A],B]]
by induction

= al[P,T,C,L, load[value[first], load[value[second],S]], cddar[A],
cons[cdr[A],B]] similarly

= al[P,T,C,L, load[(value[first] + value[second]),S], NIL, cons
[cdr[A],B]] by definition of ADD

= al[P,T,C,L, load[(value[first] + value[second]),S], cdr[A], B]
since A is NIL

giving the required result.

This is rather trivial, but at least is is rigorous. The remainder of the proof follows similar lines. It may well be preferable at this stage to reduce the lists T, C, and L to vectors in order to avoid tracing down a list for its n^{th} element. Details of this remain to be worked out. The next section indicates a basis for list-vector conversion on the more complicated program structure.

9. Translation to the Machine System.

The 1st stage of translation will involve converting a list structure into a vector. The vector will correspond to a one-level list given by

convert[L] = null[L] → NIL
atom[L] → list[L]
nconc[convert[car[L]], convert[L]]]

i.e. the elements of cdr[L] follow those of car [L] in the vector.

We include the following items and operations in the vector system. $\emptyset$ represent the empty vector.

V ⊕ x represents the vector formed by adding the item x to the "bottom" of V.

The functions top and rest reduce a vector to its components:

top [V ⊕ x] = (V = $\emptyset$ → A, top [V]) ; top [$\emptyset$] = error (1)

rest[V ⊕ x] = (V = $\emptyset$ → $\emptyset$, rest [V] ⊕ x) ; rest [$\emptyset$] = error (2)

Corresponding to these concepts, we have functions top and rest in the LISP system. If we are constructing a vector from a list and come to a sub-list A, we must keep track of the branch point in the list B, as in the previous section. We thus define a position in the list by a pair (A,B) and define

top[A,B] = null[A] - (null[B] → error, top[car[B], cdr[B]]), (3)

lisp[car[A]] → top[car[A], cons[cdr[A],B]], (4)

car[A] (5)

Here lisp is the list predicate that is true for non-empty lists. Similarly

rest[A,B] = null[A] → (null[B] → error, rest [car[B], cdr[B]]), (6)

lisp[car[A]] → rest[car[A], cons[cdr[A],B]], (7)

(cdr[A],B) (8)

The function which represents (A,B) in the vector system is

vector [A,B] = vec[A,B,∅] (9)

where vec [list A, B, vector V] = (10)

null[A] → (null[B] → V, vec [car[B], cdr [B], V]) , (11)

lisp[car[A]] → vec[car[A], cons[cdr[A],B], V] , (12)

vec[cdr[A], B, V ⊕ car [A]] (13)

We now set up equivalence between (A,B) and a vector V by

(A,B) ∼ V if null [A] ∧ null [B] ∧ V = ∅

or top [A,B] = top[V] ∧ rest[A,B] ∼ rest [V]

We show that (A,B) ∼ vector [A,B] : -

(i) null [A] ∧ null [B] => vector [A,B] = ∅ - trivial.

otherwise (ii) top [A,B] = top[vec[A,B,∅]]

proof: lemma: for V ≠ ∅, top[vec[A,B,V]] = top [V] (14)

proof: apply top to (10) - (13) with V = 0

use R.I.P. (recursion induction principle) and (1)

apply top to (10) - (13) with V = ∅

R.I.P., (14) and (1) gives RHS of (3) - (5)

and (iii)) rest [A,B] ∼ rest[vec[A,B,∅]]

proof: lemma: for V ≠ ∅, rest [vec[A,B,V]] = vec[A,B,rest[V]] (15)

proof: apply rest to (10) - (13) with V ≠ ∅

use R.I.P. and (2)

apply rest to (10) - (13) with V = ∅

R.I.P., (15) and (2) give RHS of (6) - (8)

10. Conclusion

The above work-out is essentially exploratory. It becomes immersed in a ream of trivial details all too quickly. The escape for this is to establish some elementary principles and theory for proving the "obvious".

Is the translation philosophy a practical one? The author is obviously influenced by experience with the CPL compiler at Cambridge: this is a glorified version of the above approach and is proving a very workable system. It is also interesting to note that a general pattern of compiler-writing for ALGOL-type languages is emerging in a form that is also essentially the same.

Several aspects of the proof are open to criticism. In particular, it seems unnecessary to implement the concept of the table at such an early stage. Labels are also a recurrent problem requiring further thought: to resort to parameters and an assembly routine is too easy a let-out, and it falls short of the final goal.

If the work is considered to be of sufficient potential value to the theory of computation, the author may well develop it into a more precise, complete and systematic presentation at Cambridge. Meanwhile, he acknowledges his debt to Professor McCarthy and to the Computation Center for the opportunity and encouragement to to initiate it.

REFERENCES

[1] J. McCarthy, "Towards a Mathematical Science of Computation", Proceedings of IFIP Conference '62.