A BASIS FOR A MATHEMATICAL THEORY OF COMPUTATION*

JOHN McCARTHY

1961–1963

[This 1963 paper was included in *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirshberg and published by North-Holland. An earlier version was published in 1961 in the *Proceedings of the Western Joint Computer Conference.*]

1 Introduction

Computation is sure to become one of the most important of the sciences. This is because it is the science of how machines can be made to carry out intellectual processes. We know that any intellectual process that can be carried out mechanically can be performed by a general purpose digital computer. Moreover, the limitations on what we have been able to make computers do so far clearly come far more from our weakness as programmers than from the intrinsic limitations of the machines. We hope that these limitations can be greatly reduced by developing a mathematical science of computation.

There are three established directions of mathematical research relevant to a science of computation. The first and oldest of these is numerical analysis. Unfortunately, its subject matter is too narrow to be of much help in forming a general theory, and it has only recently begun to be affected by the existence of automatic computation.

*This paper is a corrected version of the paper of the same title given at the Western Joint Computer Conference, May 1961. A tenth section discussing the relations between mathematical logic and computation has bean added.

The second relevant direction of research is the theory of computability as a branch of recursive function theory. The results of the basic work in this theory, including the existence of universal machines and the existence of unsolvable problems, have established a framework in which any theory of computation must fit. Unfortunately, the general trend of research in this field has been to establish more and better unsolvability theorems, and there has been very little attention paid to positive results and none to establishing the properties of the kinds of algorithms that are actually used. Perhaps for this reason the formalisms for describing algorithms are too cumbersome to be used to describe actual algorithms.

The third direction of mathematical research is the *theory of finite automata*. Results which use the finiteness of the number of states tend not to be very useful in dealing with present computers which have so many states that it is impossible for them to go through a substantial fraction of them in a reasonable time.

The present paper is an attempt to create a basis for a mathematical theory of computation. Before mentioning what is in the paper, we shall discuss briefly what practical results can be hoped for from a suitable mathematical theory. This paper contains direct contributions towards only a few of the goals to be mentioned, but we list additional goals in order to encourage a gold rush.

1. To develop a universal programming language. We believe that this goal has been written off prematurely by a number of people. Our opinion of the present situation is that ALGOL is on the right track but mainly lacks the ability to describe different kinds of data, that COBOL is a step up a blind alley on account of its orientation towards English which is not well suited to the formal description of procedures, and that UNCOL is an exercise in group wishful thinking. The formalism for describing computations in this paper is not presented as a candidate for a universal programming language because it lacks a number of features, mainly syntactic, which are necessary for convenient use.

2. To define a theory of the equivalence of computation processes. With such a theory we can define equivalence preserving transformations. Such transformations can be used to take an algorithm from a form in which it is easily seen to give the right answers to an equivalent form guaranteed to give the same answers but which has other advantages such as speed, economy of storage, or the incorporation of auxiliary processes.

3. To represent algorithms by symbolic expressions in such a way that significant changes in the behavior represented by the algorithms are represented by simple changes in the symbolic expressions. Programs that are supposed to learn from experience change their behavior by changing the contents of the registers that represent the modifiable aspects of their behavior. From a certain point of view, having a convenient representation of one's behavior available for modification is what is meant by consciousness.

4. To represent computers as well as computations in a formalism that permits a treatment of the relation between a computation and the computer that carries out the computation.

5. To give a quantitative theory of computation. There might be a quantitative measure of the size of a computation analogous to Shannon's measure of information. The present paper contains no information about this.

The present paper is divided into two sections. The first contains several descriptive formalisms with a few examples of their use, and the second contains what little theory we have that enables us to prove the equivalence of computations expressed in these formalisms. The formalisms treated are the following:

1. A way of describing the functions that are computable in terms of given base functions, using conditional expressions and recursive function definitions. This formalism differs from those of recursive function theory in that it is not based on the integers, strings of symbols, or any other fixed domain.

2. Computable functionals, i.e. functions with functions as arguments.

3. Non-computable functions. By adjoining quantifiers to the computable function formalism, we obtain a wider class of functions which are not *a priori* computable. However, such functions can often be shown to be equivalent to computable functions. In fact, the mathematics of computation may have, as one of its major aspects, rules which permit us to transform functions from a non-computable form into a computable form.

4. Ambiguous functions. Functions whose values are incompletely specified may be useful in proving facts about functions where certain details are irrelevant to the statement being proved.

5. A way of defining new data spaces in terms of given base spaces and of defining functions on the new spaces in terms of functions on the base spaces. Lack of such a formalism is one of the main weaknesses of ALGOL but the business data processing languages such as FLOWMATIC and COBOL have made a start in this direction, even though this start is hampered by

concessions to what the authors presume are the prejudices of business men.

The second part of the paper contains a few mathematical results about the properties of the formalisms introduced in the first part. Specifically, we describe the following:

1. The formal properties of conditional expressions.

2. A method called *recursion induction* for proving the equivalence of recursively defined functions.

3. Some relations between the formalisms introduced in this paper and other formalisms current in recursive function theory and in programming.

We hope that the reader will not be angry about the contrast between the great expectations of a mathematical theory of computation and the meager results presented in this paper.

2 Formalisms For Describing Computable Functions and Related Entities

In this part we describe a number of new formalisms for expressing computable functions and related entities. The most important section is 1, the subject matter of which is fairly well understood. The other sections give formalisms which we hope will be useful in constructing computable functions and in proving theorems about them.

2.1 Functions Computable in Terms of Given Base Functions

Suppose we are given a base collection $\mathcal{F}$ of functions (including predicates) having certain domains and ranges. In the case of the non-negative integers, we may have the successor function and the predicate of equality, and in the case of the S-expressions discussed in reference 7, we have the five basic operations. Our object is to define a class of functions $\mathcal{C}\{\mathcal{F}\}$ which we shall call *the class of functions computable in terms of* $\mathcal{F}$.

Before developing $\mathcal{C}\{\mathcal{F}\}$ formally, we wish to give an example, and in order to give the example, we first need the concept of conditional expression. In our notation a conditional expression has the form

$$(p_1 \to e_1, p_2 \to e_2, \dots, p_n \to e_n)$$

which corresponds to the ALGOL 60 reference language (12) expression

$$\textbf{if } p_1 \textbf{ then } e_1 \textbf{ else } \textbf{ if } p_2 \textbf{ then } e_2 \ \dots \ \textbf{else } \textbf{ if } p_n \textbf{ then } e_n.$$

Here $p_1, \dots, p_n$ are propositional expressions taking the values T or F standing for truth and falsity respectively.

The value of $(p_1 \to e_1, p_2 \to e_2, \dots, p_n \to e_n)$ is the value of the e corresponding to the first p that has value T. Thus

$$(4 < 3 \to 7,\ 2 > 3 \to 8,\ 2 < 3 \to 9,\ 4 < 5 \to 7) = 9.$$

Some examples of the conditional expressions for well known functions are

$$\begin{aligned} |x| &= (x < 0 \to -x, x \geq 0 \to x) \\ \delta_{ij} &= (i = j \to 1, i \neq j \to 0) \end{aligned}$$

and the triangular function whose graph is given in figure 1 is represented by the conditional expression

$$tri(x) = (x \leq -1 \to 0, x \leq 0 \to x + 1, x \leq 1 \to 1 - x, x > 1 \to 0).$$

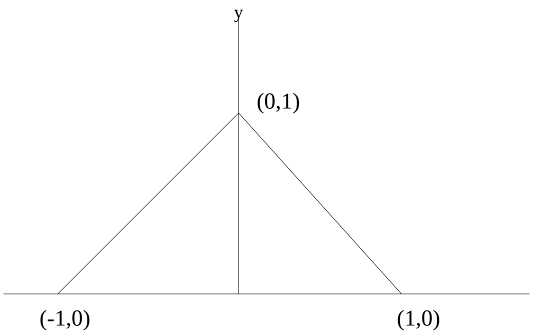

Fig. 1

Now we are ready to use conditional expressions to define functions recursively. For example, we have

$$n! = (n = 0 \to 1, n \neq 0 \to n \cdot (n-1)!)$$

Let us evaluate 2! according to this definition. We have

$$\begin{aligned}
2! &= (2 = 0 \to 1, 2 \neq 0 \to 2 \cdot (2-1)!) \\
&= 2 \cdot 1! \\
&= 2 \cdot (1 = 0 \to 1, 1 \neq 0 \to 1 \cdot (1-1)!) \\
&= 2 \cdot 1 \cdot 0! \\
&= 2 \cdot 1 \cdot (0 = 0 \to 1, 0 \neq 0 \to 0 \cdot (0-1)!) \\
&= 2 \cdot 1 \cdot 1 \\
&= 2.
\end{aligned}$$

The reader who has followed these simple examples is ready for the construction of $C\{\mathcal{F}\}$ which is a straightforward generalization of the above together with a tying up of a few loose ends.

Some notation. Let $\mathcal{F}$ be a collection (finite in the examples we shall give) of functions whose domains and ranges are certain sets. $C\{\mathcal{F}\}$ will be a class of functions involving the same sets which we shall call computable in terms of $\mathcal{F}$.

Suppose f is a function of n variables, and suppose that if we write $y = f(x_i, ..., x_n)$, each x_i takes values in the set U_i and y takes its value in the set V. It is customary to describe this situation by writing

$$f : U_1 \times U_2 \times \ldots \times U_n \to V.$$

The set $U_1 \times \cdots \times U_n$ of n-tuples $(x_1, \ldots, x_n)$ is called the *domain* of f, and the set V is called the *range* of f.

Forms and functions. In order to make properly the definitions that follow, we will distinguish between functions and expressions involving free variables. Following Church [1] the latter are called $forms$. Single letters such as f, g, h, etc. or sequences of letters such as *sin* are used to denote *functions*. Expressions such as $f(x, y)$, $f(g(x), y), x^2 + y$ are called forms. In particular we may refer to the function f defined by $f(x, y) = x^2 + y$. Our definitions will be written as though all forms involving functions were written $f(, ...,)$ although we will use expressions like $x + y$ with infixes like $+$ in examples.

Composition. Now we shall describe the ways in which new functions are defined from old. The first way may be called (generalized) *composition* and involves the use of forms. We shall use the letters $x, y, ...$ (sometimes with subscripts) for variables and will suppose that there is a notation for constants that does not make expressions ambiguous. (Thus, the decimal notation is allowed for constants when we are dealing with integers.)

The class of forms is defined recursively as follows:

(i) A variable x with an associated space U is a form, and with this form we also associate U. A constant in a space U is a form and we also associate U with this form.

(ii) If $e_1, ..., e_n$ are forms associated with the spaces $U_1, ..., U_n$ respectively, then $f(e_1, ..., e_n)$ is a form associated with the space V. Thus the form $f(g(x, y), x)$ may be built from the forms $g(x, y)$ and x and the function f.

If all the variables occurring in a form e are among $x_1, ..., x_n$, we can define a function h by writing $h(x_1, ..., x_n) = e$. We shall assume that the reader knows how to compute the values of a function defined in this way. If $f_1, ..., f_m$ are all the functions occurring in e we shall say that the function h is defined by *composition* from $f_1, ..., f_m$. The class of functions definable from given functions using only composition is narrower than the class of function computable in terms of these functions.

Partial functions. In the theory of computation it is necessary to deal with *partial functions* which are not defined for all n-tuples in their domains. Thus we have the partial function *minus*, defined by $minus(x, y) = x - y$, which is defined on those pairs (x, y) of positive integers for which x is greater than y. A function which is defined for all n-tuples in its domain is called a *total function.* We admit the limiting case of a partial function which is not defined for any n-tuples.

The n-tuples for which a function described by composition is defined is determined in an obvious way from the sets of n-tuples for which the functions entering the composition are defined. If all the functions occurring in a composition are total functions, the new function is also a total function, but the other processes for defining functions are not so kind to totality. When the word "function" is used from here on, we shall mean partial function.

Having to introduce partial functions is a nuisance, but an unavoidable one. The rules for defining computable functions sometimes give computation processes that never terminate, and when the computation process fails to terminate, the result is undefined. It is well known that there is no effective general way of deciding whether a process will terminate.

Predicates and propositional forms. The space Π of truth values whose only elements are T (for truth) and F (for falsity) has a special role in our theory. A function whose range is Π is called a *predicate*. Examples of predicates on the integers are *prime* defined by

$$\mathrm{prime}(x) = \begin{cases} \text{T if x is prime} \\ \text{F otherwise} \end{cases}$$

and *less* defined by

$$\mathrm{less}(x, y) = \begin{cases} \text{T if } x < y \\ \text{F otherwise} \end{cases}$$

We shall, of course, write $x < y$ instead of $less(x, y)$. For any space U there is a predicate eq_U of two arguments defined by

$$eq_U(x, y) = \begin{cases} \text{T if } x = y \\ \text{F otherwise} \end{cases}$$

We shall write $x = y$ instead of $eq_U(x, y)$, but some of the remarks about functions might not hold if we tried to consider equality a single predicate defined on all spaces at once.

A form with values in Π such as $x < y, x = y$, or $prime(x)$ is called a *propositional form.*

Propositional forms constructed directly from predicates such as $prime(x)$ or $x < y$ may be called *simple*. Compound propositional forms can be constructed from the simple ones by means of the propositional connectives $\wedge, \vee$, and $\sim$. We shall assume that the reader is familiar with the use of these connectives.

Conditional forms or conditional expressions. Conditional forms require a little more careful treatment than was given above in connection with the example. The value of the conditional form

$$(p_1 \rightarrow e_1, ..., p_n \rightarrow e_n)$$

is the value of the e corresponding to the first p that has value T; if all p's have value F, then the value of the conditional form is not defined. This rule is complete provided all the p's and e's have defined values, but we need to make provision for the possibility that some of the p's or e's are undefined. The rule is as follows:

If an undefined p occurs before a true p or if all p's are false or if the e corresponding to the first true p is undefined, then the form is undefined. Otherwise, the value of the form is the value of the e corresponding to the first true p.

We shall illustrate this definition by additional examples:

$$\begin{array}{l}
(2 < 1 \to 1, 2 > 1 \to 3) = 3 \\
(1 < 2 \to 4, 1 < 2 \to 3) = 4 \\
(2 < 1 \to 1, 3 < 1 \to 3) \text{ is undefined} \\
(0/0 < 1 \to 1, 1 < 2 \to 3) \text{ is undefined} \\
(1 < 2 \to 0/0, 1 < 2 \to 1) \text{ is undefined} \\
(1 < 2 \to 2, 1 < 3 \to 0/0) = 2
\end{array}$$

The truth value T can be used to simplify certain conditional forms. Thus, instead of

$$|x| = (x < 0 \to -x, x \geq 0 \to x),$$

we shall write

$$|x| = (x < 0 \to -x, T \to x).$$

The propositional connectives can be expressed in terms of conditional forms as follows:

$$\begin{aligned}
p \wedge q &= (p \to q, \mathrm{T} \to \mathrm{F}) \\
p \vee q &= (p \to \mathrm{T}, \mathrm{T} \to q) \\
\sim p &= (p \to \mathrm{F}, \mathrm{T} \to \mathrm{T}) \\
p \supset q &= (p \to q, \mathrm{T} \to \mathrm{T})
\end{aligned}$$

Considerations of truth tables show that these formulae give the same results as the usual definitions. However, in order to treat partial functions we must consider the possibility that p or q may be undefined.

Suppose that p is false and q is undefined; then according to the conditional form definition $p \wedge q$ is false and $q \wedge p$ is undefined. This unsymmetry in the propositional connectives turns out to be appropriate in the theory of computation since if a calculation of p gives F as a result q need not be computed to evaluate $p \wedge q$, but if the calculation of p does not terminate, we never get around to computing q.

It is natural to ask if a function $cond_n$ of $2n$ variables can be defined so that

$$(p_1 \rightarrow e_1, ..., p_n \rightarrow e_n) = cond_n(p, ..., p_n, e_1, ..., e_n).$$

This is not possible unless we extend our notion of function because normally one requires all the arguments of a function to be given before the function is computed. However, as we shall shortly see, it is important that a conditional form be considered defined when, for example, p_1 is true and e_1 is defined and all the other p's and e's are undefined. The required extension of the concept of function would have the property that functions of several variables could no longer be identified with one-variable functions defined on product spaces. We shall not pursue this possibility further here.

We now want to extend our notion of forms to include conditional forms. Suppose $p_1, ..., p_n$ are forms associated with the space of truth values and $e_1, ..., e_n$ are forms each of which is associated with the space V. Suppose further that each variable x_i occurring in $p_1, ..., p_n$ and $e_1, ..., e_n$ is associated with the space U. Then $(p_1 \rightarrow e_1, ..., p_n \rightarrow e_n)$ is a form associated with V.

We believe that conditional forms will eventually come to be generally used in mathematics whenever functions are defined by considering cases. Their introduction is the same kind of innovation as vector notation. Nothing can be proved with them that could not also be proved without them. However, their formal properties, which will be discussed later, will reduce many case-analysis verbal arguments to calculation.

Definition of functions by recursion. The definition

$$n! = (n = 0 \rightarrow 1, \mathrm{T} \rightarrow n \cdot (n - 1)!)$$

is an example of definition by recursion. Consider the computation of 0!

$$0! = (0 = 0 \rightarrow 1, \mathrm{T} \rightarrow 0 \cdot (0 - 1)!) = 1.$$

We now see that it is important to provide that the conditional form be defined even if a term beyond the one that gives the value is undefined. In this case (0 - 1)! is undefined.

Note also that if we consider a wider domain than the non-negative integers, $n!$ as defined above becomes a partial function, since unless n is a non-negative integer, the recursion process does not terminate.

In general, we can either define single functions by recursion or define several functions together by simultaneous recursion, the former being a particular case of the latter.

To define simultaneously functions $f_1, ..., f_k$, we write equations

$$f_1(x_1, ..., x_n) = e_1$$
$$\vdots$$
$$f_k(x_1, ..., x_n) = e_k$$

The expressions $e_1, ..., e_k$ must contain only known functions and the functions $f_1, ..., f_k$. Suppose that the ranges of the functions are to be $V_1, ..., V_k$ respectively; then we further require that the expressions $e_1, ..., e_k$ be associated with these spaces respectively, given that within $e_1, ..., e_k$ the f's are taken as having the corresponding V's as ranges. This is a consistency condition.

$f_i(x_i, ..., x_k)$ is to be evaluated for given values of the x's as follows.

1. If e_i is a conditional form then the p's are to be evaluated in the prescribed order stopping when a true p and the corresponding e have been evaluated.

2. If e_i has the form $g(e_1^*, ..., e_m^*)$, then $e_1^*, ..., e_m^*$ are to be evaluated and then the function g applied.

3. If any expression $f_i(e_1^*, ..., e_n^*)$ occurs it is to be evaluated from the defining equation.

4. Any subexpressions of e_i that have to be evaluated are evaluated according to the same rules.

5. Variables occurring as subexpressions are evaluated by giving them the assigned values.

There is no guarantee that the evaluation process will terminate in any given case. If for particular arguments the process does not terminate, then the function is undefined for these arguments. If the function f_i occurs in the expression e_i, then the possibility of termination depends on the presence of conditional expressions in the e_i's.

The class of functions $C\{\mathcal{F}\}$ computable in terms of the given base functions $\mathcal{F}$ is defined to consist of the functions which can be defined by repeated applications of the above recursive definition process.

2.2 Recursive Functions of the Integers

In Reference 7 we develop the recursive functions of a class of symbolic expressions in terms of the conditional expression and recursive function formalism.

As an example of the use of recursive function definitions, we shall give recursive definitions of a number of functions over the integers. We do this for three reasons: to help the reader familiarize himself with recursive definition, to show how much simpler in practice our methods of recursive definition are than either Turing machines or Kleene's formalism, and to prove that any partial recursive function (Kleene) on the non-negative integers is in $C\{\mathcal{F}\}$ where $\mathcal{F}$ contains only the successor function and the predicate equality.

Let I be the set of non-negative integers $\{0,1,2,...\}$ and denote the successor of an integer n by n' and denote the equality of integers n_1 and n_2 by $n_1 = n_2$. If we define functions *succ* and *eq* by

$$succ(n) = n'$$

$$eq(n_1, n_2) = \begin{cases} \mathrm{T} \text{ if } n_1 = n_2 \\ \mathrm{F} \text{ if } n_1 \neq n_2 \end{cases}$$

then we write $\mathcal{F} = \{succ, eq\}$. We are interested in $C\{\mathcal{F}\}$. Clearly all functions in $C\{\mathcal{F}\}$ will have either integers or truth values as values.

First we define the predecessor function *pred*(not defined for $n = 0$) by

$$pred(n) = pred2(n, 0)$$

$$pred2(n, m) = (m' = n \rightarrow m, \mathrm{T} \rightarrow pred2(n, m')).$$

We shall denote $pred(n)$ by n^-.

Now we define the sum

$$m + n = (n = 0 \rightarrow m, \mathrm{T} \rightarrow m' + n^-),$$

the product

$$mn = (n = 0 \rightarrow 0, \mathrm{T} \rightarrow m + mn^-),$$

the difference

$$m - n = (n = 0 \rightarrow m, \mathrm{T} \rightarrow m^- - n^-)$$

which is defined only for $m \geq n$. The inequality predicate $m \leq n$ is defined by

$$m \leq n = (m = 0) \vee (\sim (n = 0) \wedge (m^- \leq n^-)).$$

The strict inequality $m < n$ is defined by

$$m < n = (m \leq n) \wedge \sim (m = n).$$

The integer valued quotient m/n is defined by

$$m/n = (m < n \rightarrow 0, \mathrm{T} \rightarrow ((m - n)/n)').$$

The remainder on dividing m by n is defined by

$$rem(m/n) = (m < n \rightarrow m, \mathrm{T} \rightarrow rem((m - n)/n)),$$

and the divisibility of a number n by a number m,

$$m|n = (n = 0) \vee ((n \geq m) \wedge (m|(n - m))).$$

The primeness of a number is defined by

$$prime(n) = (n \neq 0) \wedge (n \neq 1) \wedge prime2(n, 2)$$

where

$$prime2(m, n) = (m = n) \vee (\sim (m|n) \wedge prime2(n, m')).$$

The Euclidean algorithm defines the greatest common divisor, and we write

$$gcd(m, n) = (m > n \rightarrow gcd(n, m), rem(n/m) = 0 \rightarrow m, \mathrm{T} \rightarrow gcd(rem(n/m), m))$$

and we can define Euler's φ-function by

$$\varphi(n) = \varphi_2(n, n)$$

where

$$\varphi_2(n, m) = (m = 1 \to 1, gcd(n, m) = 1 \to \varphi_2(n, m^-)', \mathrm{T} \to \varphi_2(n, m^-)).$$

$\varphi(n)$ is the number of numbers less than n and relatively prime to n.

The above shows that our form of recursion is a convenient way of defining arithmetical functions. We shall see how some of the properties of the arithmetical functions can conveniently be derived in this formalism in a later section.

2.3 Computable Functionals

The formalism previously described enables us to define functions that have functions as arguments. For example,

$$\sum_{i=m}^{n} a_i$$

can be regarded as a function of the numbers m and n and the sequence $\{a_i\}$. If we regard the sequence as a function f we can write the recursive definition

$$sum(m, n, f) = (m > n \to 0, \mathrm{T} \to f(m) + sum(m + 1, n, f))$$

or in terms of the conventional notation

$$\sum_{i=m}^{n} f(i) = (m > n \to 0, \mathrm{T} \to f(m) + \sum_{i=m+1}^{n} f(i)).$$

Functions with functions as arguments are called *functionals*.

Another example is the functional $least(p)$ which gives the least integer n such that $p(n)$ for a predicate p. We have

$$least(p) = least2(p, 0)$$

where

$$least2(p, n) = (p(n) \to n, \mathrm{T} \to least2(p, n + 1)).$$

In order to use functionals it is convenient to have a notation for naming functions. We use Church's [1] lambda notation. Suppose we have a function f defined by an equation $f(x_1, ..., x_n) = e$ where e is some expression in $x_1, ..., x_n$. The name of this function is $\lambda((x_1, ..., x_n), e)$. For example, the name of the function f defined by $f(x, y) = x^2 + y$ is $\lambda((x, y), x^2 + y)$.

Thus we have

$$\lambda((x, y), x^2 + y)(3, 4) = 13,$$

but

$$\lambda((y, x), x^2 + y)(3, 4) = 19.$$

The variables occurring in a λ definition are dummy or bound variables and can be replaced by others without changing the function provided the replacement is done consistently. For example, the expressions

$$\lambda((x, y), x^2 + y),$$
$$\lambda((u, v), u^2 + v),$$

and

$$\lambda((y, x), y^2 + x)$$

all represent the same function.

In the notation $\sum_{i=1}^{n} i^2$ is represented by $sum(1, n, \lambda((i), i^2))$ and the least integer n for which $n^2 > 50$ is represented by

$$least(\lambda((n), n^2 > 50)).$$

When the functions with which we are dealing are defined recursively, a difficulty arises. For example, consider $factorial$ defined by

$$factorial(n) = (n = 0 \rightarrow 1, \mathrm{T} \rightarrow n \cdot factorial(n - 1)).$$

The expression

$$\lambda((n), (n = 0 \rightarrow 1, \mathrm{T} \rightarrow n \cdot factorial(n - 1)))$$

cannot serve as a name for this function because it is not clear that the occurrence of "factorial" in the expression refers to the function defined by the expression as a whole. Therefore, for recursive functions we adopt an additional convention, Namely,

$$label(f, \lambda((x_1, ..., x_n), e))$$

stands for the function f defined by the equation

$$f(x_1, ..., x_n) = e$$

where any occurrences of the function letter f within e stand for the function being defined. The letter f is a dummy variable. The factorial function then has the name

$$label(factorial, \lambda((n), (n = 0 \to 1, \mathrm{T} \to n \cdot factorial(n-1)))),$$

and since $factorial$ and n are dummy variables the expression

$$label(g, \lambda((r), (r = 0 \to 1, \mathrm{T} \to r \cdot g(r-1))))$$

represents the same function.

If we start with a base domain for our variables, it is possible to consider a hierarchy of functionals. At level 1 we have functions whose arguments are in the base domain. At level 2 we have functionals taking functions of level 1 as arguments. At level 3 are functionals taking functionals of level 2 as arguments, etc. Actually functionals of several variables can be of mixed type.

However, this hierarchy does not exhaust the possibilities, and if we allow functions which can take themselves as arguments we can eliminate the use of *label* in naming recursive functions. Suppose that we have a function f defined by

$$f(x) = \mathcal{E}(x, f)$$

where $\mathcal{E}(x, f)$ is some expression in x and the function variable f. This function can be named

$$label(f, \lambda((x), \mathcal{E}(x, f))).$$

However, suppose we define a function g by

$$g(x, \varphi) = \mathcal{E}(x, \lambda((x), \varphi(x, \varphi)))$$

or

$$g = \lambda((x, \varphi), \mathcal{E}(x, \lambda((x), \varphi(x, \varphi)))).$$

We then have

$$f(x) = g(x, g)$$

since $g(x, g)$ satisfies the equation

$$g(x, g) = \mathcal{E}(x, \lambda((x), g(x, g))).$$

Now we can write f as

$$f = \lambda((x), \lambda((y, \varphi), \mathcal{E}(y, \lambda((u), \varphi(u, \varphi))))(x, \lambda((y, \varphi), \mathcal{E}(y, \lambda((u), \varphi(u, \varphi)))))).$$

This eliminates *label* at what seems to be an excessive cost. Namely, the expression gets quite complicated and we must admit functionals capable of taking themselves as arguments. These escape our orderly hierarchy of functionals.

2.4 Non-Computable Functions and Functionals

It might be supposed that in a mathematical theory of computation one need only consider computable functions. However, mathematical physics is carried out in terms of real valued functions which we not computable but only approximable by computable functions.

We shall consider several successive extensions of the class $C\{\mathcal{F}\}$. First we adjoin the universal quantifier $\forall$ to the operations used to define new functions. Suppose e is a form in a variable x and other variables associated with the space Π of truth values. Then

$$\forall((x), e)$$

is a new form in the remaining variables also associated with Π. $\forall((x), e)$ has the value T for given values of the remaining variables if for all values of x, e has the value T. $\forall((x), e)$ has the value F if for at least one value of x, e has the value F. In the remaining case, i.e. for some values of x, e has the value T and for all others e is undefined, $\forall((x), e)$ is undefined.

If we allow the use of the universal quantifier to form new propositional forms for use in conditional forms, we get a class of functions $Ha\{\mathcal{F}\}$ which may well be called the class of functions hyper-arithmetic over $\mathcal{F}$ since in the case where $\mathcal{F} = \{successor, equality\}$ on the integers, $Ha\{\mathcal{F}\}$ consists of Kleene's hyper-arithmetic functions.

Our next step is to allow the description operator $\imath$. $\imath((x), p(x))$ stands for the unique x such that $p(x)$ is true. Unless there is such an x and it is unique, $\imath((x), p(x))$ is undefined. In the case of the integers $\imath((x), p(x))$ can be defined in terms of the universal quantifier using conditional expressions, but this does not seem to be the case in domains which are not effectively enumerable, and one may not wish to do so in domains where enumeration is unnatural.

The next step is to allow quantification over functions. This gets us to Kleene's [5] analytic hierarchy and presumably allows the functions used in analysis. Two facts are worth noting. First $\forall((f), \varphi(f))$ refers to all functions on the domain and not just the computable ones. If we restrict quantification to computable functions, we get different results. Secondly, if we allow functions which can take themselves as arguments, it is difficult to assign a meaning to the quantification. In fact, we are apparently confronted with the paradoxes of naive set theory.

2.5 Ambiguous Functions

Ambiguous functions are not really functions. For each prescription of values to the arguments the ambiguous function has a collection of possible values. An example of an ambiguous function is $less(n)$ defined for all positive integer values of n. Every non-negative integer less than n is a possible value of $less(n)$. First we define a basic ambiguity operator $amb(x, y)$ whose possible values are x and y when both are defined: otherwise, whichever is defined. Now we can define $less(n)$ by

$$less(n) = amb(n-1, less(n-1)).$$

$less(n)$ has the property that if we define

$$ult(n) = (n = 0 \to 0, \mathrm{T} \to ult(less(n)))$$

then

$$\forall((n), ult(n) = 0) = T.$$

There are a number of important kinds of mathematical arguments whose convenient formalization may involve ambiguous functions. In order to give an example, we need two definitions.

If f and g are two ambiguous functions, we shall say that f is a *descendant* of g if for each x every possible value of $f(x)$ is also a possible value of $g(x)$.

Secondly, we shall say that a property of ambiguous functions is *hereditary* if whenever it is possessed by a function g it is also possessed by all descendants of g. The property that iteration of an integer valued function eventually gives 0 is hereditary, and the function *less* has this property. So, therefore, do all its descendants. Therefore any integer-function g satisfying $g(0) = 0$ and $n > 0 \supset g(n) < n$ has the property that $g^*(n) = (n = 0 \rightarrow 0, \mathrm{T} \rightarrow g^*(g(n)))$ is identically 0 since g is a descendant of *less*. Thus any function, however complicated, which always reduces a number will if iterated sufficiently always give 0.

This example is one of our reasons for hoping that ambiguous functions will turn out to be useful.

With just the operation *amb* defined above adjoined to those used to generate $C\{\mathcal{F}\}$, we can extend $\mathcal{F}$ to the class $C^*\{\mathcal{F}\}$ which may be called the *computably ambiguous functions*. A wider class of ambiguous functions is formed using the operator $Am(x, \pi(x))$ whose values are all x's satisfying $\pi(x)$.

2.6 Recursive Definitions of Sets

In the previous sections on recursive definition of functions the domains and ranges of the basic functions were prescribed and the defined functions had the same domains and ranges.

In this section we shall consider the definition of new sets and the basic functions on them. First we shall consider some operations whereby new sets can be defined.

1. The Cartesian product $A \times B$ of two sets A and B is the set of all ordered pairs $(a \cdot b)$ with $a \epsilon A$ and $b \epsilon B$. If A and B are finite sets and $n(A)$ and $n(B)$ denote the numbers of members of A and B respectively then $n(A \times B) = n(A) \cdot n(B)$.

Associated with the pair of sets (A, B) are two *canonical mappings:*

$$\pi_{A,B} : A \times B \rightarrow A \text{ defined by } \pi_{A,B}((a \cdot b)) = a$$
$$\varrho_{A,B} : A \times B \rightarrow B \text{ defined by } \varrho_{A,B}((a \cdot b)) = b.$$

The word "canonical" refers to the fact that $\pi_{A,B}$ and $\varrho_{A,B}$ are defined by the sets A and B and do not depend on knowing anything about the members of A and B.

The next canonical function γ is a function of two variables $\gamma_{A,B} : A, B \to A \times B$ defined by

$$\gamma_{A,B}\,(a, b) = (a \cdot b).$$

For some purposes functions of two variables, x from A and y from B, can be identified with functions of one variable defined on $A \times B$.

2. The direct union $A \oplus B$ of the sets A and B is the union of two non-intersecting sets one of which is in 1-1 correspondence with A and the other with B. If A and B are finite, then $n(A \oplus B) = n(A) + n(B)$ even if A and B intersect. The elements of $A \oplus B$ may be written as elements of A or B subscripted with the set from which they come, i.e. a_A or b_B.

The canonical mappings associated with the direct union $A \oplus B$ are

$i_{A,B} : A \to A \oplus B$ defined by $i_{A,B}\,(a) = a_A$,
$j_{A,B} : B \to A \oplus B$ defined by $j_{A,B}\,(b) = b_B$,
$p_{A,B} : A \oplus B \to \Pi$ defined by $p_{A,B}\,(x) = \mathrm{T}$ if and only if x comes from A,
$q_{A,B} : A \oplus B \to \Pi$ defined by $q_{A,B}\,(x) = \mathrm{T}$ if and only if x comes from B.

There are two canonical partial functions $r_{A,B}$ and $s_{A,B}$. $r_{A,B} \colon A \oplus B \to A$ is defined only for elements coming from A and satisfies $r_{A,B}(i_{A,B}\,(a)) = a$. Similarly, $s_{A,B} : A \oplus B \to B$ satisfies $s_{A,B}\,(j_{A,B}\,(b)) = b$.

3. The power set A^B is the set of all mappings $f \colon B \to A$. The canonical mapping $\alpha_{A,B} : A^B \times B \to A$ is defined by $\alpha_{A,B}\,(f, b) = f(b)$.

Canonical mappings. We will not regard the sets $A \times (B \times C)$ and $(A \times B) \times C$ as the same, but there is a canonical 1-1 mapping between them,

$$g_{A,B,C} \colon (A \times B) \times C \to A \times (B \times C)$$

defined by

$$g_{A,B,C}(u) = \gamma_{A,B\times C}(\pi_{A,B}(\pi_{A\times B,C}(u)), \gamma_{B,C}(\varrho_{A,B}(\pi_{A\times B,C}(u)), \varrho_{A\times B,C}(u))).$$

We shall write

$$(A \times B) \times C \simeq A \times (B \times C)$$

to express the fact that these sets are canonically isomorphic.

Other canonical isomorphisms are

1. $t_{A,B}\colon A \times B \to B \times A$ defined by $t(u) = \gamma_{B,A}(\varrho_{A,B}(u), \pi_{A,B}(u))$
2. $d_1\colon A \times (B \oplus C) \to A \times B \oplus A \times C$
3. $a_2\colon (A \oplus B) \oplus C \to A \oplus (B \oplus C)$
4. $d_2\colon A^C \times B^C \to (A \times B)^C$
5. $d_3\colon A^B \times A^C \to A^{B \oplus C}$
6. $s_1\colon (A^B)C \to A^{B \times C}$

We shall denote the null set (containing no elements) by 0 and the set consisting of the integers from 1 to n by n. We have

$$\begin{aligned}
&A \oplus 0 \simeq A \\
&A \times 0 \simeq 0 \\
&A \times 1 \simeq A \\
&A \times 2 \simeq A \oplus A (n \text{ terms, associate to left by convention}) \\
&A^0 \simeq 1 \qquad (\text{by convention}) \\
&A^1 \simeq A \\
&A^n \simeq A \times ... \times A (n \text{ terms, associate to left by convention})
\end{aligned}$$

Suppose we write the recursive equation

$$S = \{\Lambda\} \oplus A \times S.$$

We can interpret this as defining the set of sequences of elements of A as follows:

1. Interpret Λ as denoting the null sequence. Then the null sequence (strictly an image of it) is an element of S.

2. Since a pair consisting of an element of A and an element of S is an element of S, a pair (a, Λ) is an element of S. So, then, are

$$(a_1 \cdot (a_2 \cdot \Lambda)) \text{ and } (a_1 \cdot (a_2 \cdot (a_3 \cdot \Lambda))) \text{ etc.}$$

Thus S consists of all sequences of elements of A including the null sequence.

Suppose we substitute $\{\Lambda\} \oplus A \times S$ for S in the right side of $S = \{\Lambda\} \oplus A \times S$. We get

$$S = \{\Lambda\} \oplus A \times (\{\Lambda\} \oplus A \times S).$$

If we again substitute for S and expand by the distributive law expressed in equation (2) above we get

$$S = \{\Lambda\} \oplus A \times \{\Lambda\} \oplus A \times A \times \{\Lambda\} \oplus ...$$

which, if we now denote the set $\{\Lambda\}$ by 1, becomes

$$S = 1 \oplus A \oplus A^2 \oplus A^3 \oplus ...$$

which is another way of writing the set of sequences. We shall denote the set of sequences of elements of A by $seq(A)$.

We can also derive this relation by writing $S = 1 \oplus A \times S$ and solving formally for S, getting $S = 1/(1 - A)$ which we expand in geometric series to get $S = 1 \oplus A \oplus A^2 \oplus ...$ just as before.

Another useful recursive construction is

$$S = A \oplus S \times S.$$

Its elements have the forms a or $(a_1 \cdot a_2)$ or $((a_1 \cdot a_2) \cdot a_3)$ or $(a_1 \cdot (a_2 \cdot a_3))$ etc. Thus we have the set of S-expressions on the alphabet A which we may denote by $sexp(A)$. This set is the subject matter of Reference 7, and the following paragraph refers to this paper.

When sets are formed by this kind of recursive definition, the canonical mappings associated with the direct sum and Cartesian product operations have significance. Consider, for example, $sexp(A)$.

We can define the basic operations of Lisp, i.e. $atom, eq, car, cdr$ and $cons$ by the equations

$$atom(x) = p_{A,S \times S}(x)$$

$$eq(x, y) = (i_{A,S \times S}(x) = i_{A,S \times S}(y))$$

assuming that equality is defined on the space A.

$$car(x) = n_{S,S}(s_{A,S \times S}(x))$$

$$cdr(x) = \varrho_{S,S}(s_{A,S\times S}(x))$$

$$cons(x,y) = j_{A,S\times S}(\gamma_{S,S}(x,y))$$

Definition of the set of integers. Let 0 denote the null set as before. We can define the set of integers I by

$$I = \{0\} \oplus \{0\} \times I.$$

Its elements are then

$$0, (0\cdot 0), (0\cdot(0\cdot 0)), \text{etc.}$$

which we shall denote by 0,1,2,3 etc. The successor and predecessor functions are then definable in terms of the canonical operations of the defining equation. We have

$$succ(n) = \gamma(0,n)$$

$$pred(n) = \varrho(s(n)).$$

3 Properties of Computable Functions

The first part of this paper was solely concerned with presenting descriptive formalisms. In this part we shall establish a few of the properties of the entities we previously introduced. The most important section is section 8 which deals with recursion induction.

3.1 Formal Properties of Conditional Forms

The theory of conditional expressions corresponds to analysis by cases in mathematics and is only a mild generalization of propositional calculus.

We start by considering expressions called generalized Boolean forms (gbf) formed as follows:

1. Variables are divided into propositional variables p, q, r, etc. and general variables x, y, z, etc.

2. We shall write $(p \to x, y)$ for $(p \to x, \mathrm{T} \to y)$. $(p \to x, y)$ is called an elementary conditional form (ecf) of which p, x, and y are called the *premiss, conclusion* and the *alternative*, respectively.[1]

3. A variable is a gbf, and if it is a propositional variable it is called a propositional form (pf).

4. If π is a pf and α and β are gbfs, then $(\pi \to \alpha, \beta)$ is a gbf. If, in addition, α and β are pfs, so is $(\pi \to \alpha, \beta)$.

The value of a gbf α for given values (T, F or undefined) of the propositional variables will be T or F in case α is a pf or a general variable otherwise. This value is determined for a gbf $(\pi \to \alpha, \beta)$ according to the table

value(π)	value($(\pi \to \alpha, \beta)$)
T	value(α)
F	value(β)
undefined	undefined

We shall say that two gbfs are strongly equivalent if they have the same value for all values of the propositional variables in them including the case of undefined propositional variables. They are weakly equivalent if they have the same values for all values of the propositional variables when these are restricted to F and T.

The equivalence of gbfs can be tested by a method of truth tables identical to that of propositional calculus. The table for $((p \to q, r) \to a, b)$ and $(p \to (q \to a, b), (r \to a, b))$ is given on the foregoing page.

[1] 1996: I adopted Alonzo Church's spelling for the premise of an argument. It was not adopted generally, and I later gave up.

p q r	$(p \to q, r)$	$((p \to q, r) \to a, b)$	$(q \to a, b)$	$(r \to a, b)$	$(p \to (q \to a, b), (r \to a, b))$
T T T	T	a	a	a	a
T T F	T	a	a	a	a
T T u	T	a	a	u	a
T F T	F	b	b	a	b
T F F	F	b	b	b	b
T F u	F	b	b	u	b
T u T	u	u	u	a	u
T u F	u	u	u	b	u
T u u	u	u	u	u	u
F T T	T	a	a	a	a
F T F	F	b	a	b	b
F T u	u	u	a	u	u
F F T	T	a	b	a	a
F F F	F	b	b	b	b
F F u	u	u	b	u	u
F u T	T	a	u	a	a
F u F	F	b	u	b	b
F u u	u	u	u	u	u
u T T	u	u	a	a	u
u T F	u	u	a	b	u
u T u	u	u	a	u	u
u F T	u	u	b	a	u
u F F	u	u	b	a	u
u F u	u	u	b	u	u
u u T	u	u	u	a	u
u u F	u	u	u	b	u
u u u	u	u	u	u	u

According to the table, $((p \to q, r) \to a, b)$ and $(p \to (q \to a, b), (r \to a, b))$ are strongly equivalent.

For weak equivalence the u case can be left out of the table.

Consider the table,

p q	$(q \to a, b)$	$(q \to c, d)$	$(p \to (q \to a, b), (q \to c, d))$	$(p \to a, c)$	$(p \to b, d)$	$(q \to (p \to a, c), (p \to b, d))$
T T	a	c	a	a	b	a
T F	b	d	b	a	b	b
F T	a	c	c	c	d	c
F F	b	d	d	c	d	d

which proves that $(p \to (q \to a, b), (q \to c, d))$ and $(q \to (p \to a, c), (p \to b, d))$ are weakly equivalent. They are also strongly equivalent. We shall write $\equiv_s$ and $\equiv_w$ for the relations of strong and weak equivalence.

There are two rules whereby an equivalence can be used to generate other equivalences.

1. If $\alpha \equiv \beta$ and $\alpha_1 \equiv B_1$ is the result of substituting any gbf for any variable in $\alpha \equiv \beta$, then $\alpha_1 \equiv \beta_1$. This is called the rule of substitution.

2. If $\alpha \equiv \beta$ and α is subexpression of γ and δ is the result of replacing an occurrence of α in γ by an occurrence of β, then $\gamma \equiv \delta$. This is called the rule of replacement.

These rules are applicable to either strong or weak equivalence and in fact to much more general situations.

Weak equivalence corresponds more closely to equivalence of truth functions in propositional calculus than does strong equivalence.

Consider the equations

1) $(p \to a, a) \equiv_w a$
2) $(T \to a, b) \equiv_s a$
3) $(F \to a, b) \equiv_s b$
4) $(p \to \mathrm{T}, \mathrm{F}) \equiv_s p$
5) $(p \to (p \to a, b), c) \equiv_s (p \to a, c)$
6) $(p \to a, (p \to b, c) \equiv_s (p \to a, c)$
7) $((p \to q, r) \to a, b) \equiv_s (p \to (q \to a, b), (r \to a, b))$
8) $((p \to (q \to a, b), (q \to c, d)) \equiv_s (q \to (p \to a, c), (p \to b, d))$

All are strong equivalence except the first, and all can be proved by truth tables.

These eight equations can be used as axioms to transform any gbf into any weakly equivalent one using substitution and replacement.
In fact, they can be used to transform any gbf into a canonical form. This canonical form is the following. Let $p_1, ..., p_n$ be the variables of the gbf a taken in an arbitrary order. Then a can be transformed into the form

$$(p_1 \rightarrow a_0, a_1)$$

where each a_i has the form

$$a_i = (p_2 \rightarrow a_{i0}, a_{i1})$$

and in general for each $k = 1, ..., n-1$

$$a_{i_1 \ldots i_k} = (p_{k+1} \rightarrow a_{i_1 \ldots i_k 0}, a_{i_1 \ldots i_k 1})$$

and each $a_{i_1 \ldots i_n}$ is a truth value or a general variable.
For example, the canonical form of

$$((p \rightarrow q, r) \rightarrow a, b)$$

with the variables taken in the order r, q, p is

$$(r \rightarrow (q \rightarrow (p \rightarrow a, a), (p \rightarrow b, a)), (q \rightarrow (p \rightarrow a, b), (p \rightarrow b, b))).$$

In this canonical form, the 2^n cases of the truth or falsity of $p_1, ..., p_n$ are explicitly exhibited.

An expression may be transformed into canonical form as follows:

1) Axiom 7 is used repeatedly until in every subexpression the π in $(\pi \rightarrow \alpha, \beta)$ consists of a single propositional variable.

2) The variable p_1 is moved to the front by repeated application of axiom 8. There are three cases: $(q \rightarrow (p_1 \rightarrow a, b), p_1 \rightarrow c, d))$ to which axiom 8 is directly applicable; $(q \rightarrow a, (p_1 \rightarrow c, d))$ where axiom 8 becomes applicable after axiom 1 is used to make it $(q \rightarrow (p_1 \rightarrow a, a), (p_1 \rightarrow c, b))$; the case $(q \rightarrow (p_1 \rightarrow a, b), c)$ which is handled in a manner similar to that of case 2.

Once the main expression has the form $(p_1 \rightarrow \alpha, \beta)$ we move any p_1's which occur in α and β to the front and eliminate them using axioms 5 and 6. We then bring p_2 to the front of α and β using axiom 1 if necessary to guarantee at least one occurrence of p_2 in each of α and β. The process is continued until the canonical form is achieved.

There is also a canonical form for strong equivalence. Any gbf a is strongly equivalent to one of the form $(p_1 \to \alpha, \beta)$, where α and β do not contain p_1 and are themselves in canonical form. However, the variable p_1 may not be chosen arbitrarily but must be an *inevitable* propositional variable of the original gbf and can be chosen to be any inevitable variable. An inevitable variable of a gbf $(\pi \to \alpha, \beta)$ is defined to be either the first propositional variable or else an inevitable variable of both α and β. Thus p and q are the inevitable variables of

$$(p \to (r \to (q \to a, b), (q \to c, d)), (q \to e, f)).$$

A gbf a may be put in strong canonical form as follows:

1) Use axiom 7 to get all premisses as propositional variables.

2) Choose any inevitable variable, say p_1, and put a in the form $(p_1 \to \alpha, \beta)$ by using axiom 8.

3) The next step is to eliminate occurrences of p_1 in α and β. This can be done by the general rule that in any ecf occurrences of the premiss in the conclusion can be replaced by T and occurrences in the alternative by F. However, if we wish to use substitution and replacement on formulas we need the additional axioms

(9) $(p \to (q \to a, b), c) \equiv_s (p \to (q \to (p \to a, a), (p \to b, b)), c)$

and

(10) $(p \to a, (q \to b, c)) \equiv_s (p \to a, (q \to (p \to b, b), (p \to c, c)))$.

Suppose there is an occurrence of p_1 in the conclusion; we want to replace it by T. To do this, we use axioms 9 and 10 to move in a p_1 until the objectionable p_1 occurs as the inner p_1 of one of the forms

$$p_1 \to (p_1 \to a, b), c)$$

or

$$p_1 \to a, (p_1 \to b, c)).$$

In either case, the objectionable p_1 can be removed by axiom 5 or 6, and the p_1's that were moved in can be moved out again.

Thus we have $(p_1 \to \alpha, \beta)$ with p_1 missing from α and β.

4) Inevitable variables are then brought to the front of α and β and so forth.

Two gbfs are equivalent (weakly or strongly) if and only if they have the same (weak or strong) canonical form. One way this is easy to prove; if two gbfs have the same canonical form they can be transformed into each other via the canonical form. Suppose two gbfs have different weak canonical forms when the variables are taken in the same order. Then values can be chosen for the p's giving different values for the form proving non-equivalence. In the strong case, suppose that two gbfs do not have the same inevitable propositional variables. Let p be inevitable in a but not in b. Then if the other variables are assigned suitable values b will be defined with p undefined. However, a will be undefined since p is inevitable in a which proves non-equivalence. Therefore, strongly equivalent gbfs have the same inevitable variables, so let one of them be put in front of both gbfs. The process is then repeated in the conclusion and alternative etc.

The general conditional form

$$(p_1 \to e_1, ..., p_n \to e_n)$$

can be regarded as having the form

$$(p_1 \to e_1, (p_2 \to e_2, ..., (p_n \to e_n, u), ...))$$

where u is a special undefined variable and their properties can be derived from those of gbf's.

The relation of functions to conditional forms is given by the distributive law

$$f(x_1, ..., x_{i-1}, (p_1 \to e_1, ..., p_n \to e_n), x_{i+1}, ..., x_k) =$$
$$(p_1 \to f(x_1, ..., x_{i-1}, e_1, x_{i+1}, ..., x_k), ..., p_n \to f(x_1, ..., x_{i-1}, e_n, x_{i+1}, ..., x_k)).$$

The rule of replacement can be extended in the case of conditional expressions. Suppose α is an occurrence of a subexpression of an expression β.

We define a certain propositional expression π called the premiss of α in β as follows:

1) The premiss of α in α is T

2) The premiss of α in $f(x_1, ..., x_i, ..., x_n)$ where α is part of x_i is the premiss of α in x_i.

3) If α occurs in e_1 and the premiss of α in e_i is π, then the premiss of α in $(p_1 \rightarrow e_1, ..., p_i \rightarrow e_i, ..., p_n \rightarrow e_n)$ is $(\sim p_1 \wedge ... \wedge \sim p_{i-1}) \wedge p_i \wedge \pi$.

4) If α occurs in p_i and the premiss of α in p_i is π, then the premiss of α in $(p_1 \rightarrow e_1, ..., p_i \rightarrow e_i, ..., p_n \rightarrow e_n)$ is $\sim p_1 \wedge ... \wedge \sim p_{i-1} \wedge \pi$.

The extension of the rule of replacement is that an occurrence of α in β may be replaced by α' if $(\pi \rightarrow \alpha) \equiv_s (\pi \rightarrow \alpha')$ where π is the premiss of α in β. Thus in a subcase one needs only prove equivalence under the premiss of the subcase.

3.2 Recursion Induction

Suppose a function f is defined recursively by

$$(1) \qquad f(x_1, ..., x_n) = \mathcal{E}\ \{x_1, ..., x_n, f\}$$

where $\mathcal{E}$ is an expression that in general contains f. Suppose that $\mathcal{A}$ is the set of n-tuples $(x_1, ..., x_n)$ for which f is defined. Now let g and h be two other functions with the same *domain* as f and which are defined for all n-tuples in $\mathcal{A}$. Suppose further that g and h satisfy the equation which defined f. We assert that

$$g(x_1, ..., x_n) = h(x_1, ..., x_n)$$

for all $(x_1, ..., x_n)$ in $\mathcal{A}$. This is so, simply because equation (1) uniquely determines the value that any function satisfying it has for arguments in $\mathcal{A}$ which in turn follows from the fact that (1) can be used to compute $f(x_1, ..., x_n)$ for $(x_1, ..., x_n)$ in $\mathcal{A}$.

We shall call this method of proving two functions equivalent by the name of *recursion induction.*

We shall develop some of the properties of the elementary functions of integers in order to illustrate proof by recursion induction. We recall the definitions

$$m + n = (n = 0 \to m, \mathrm{T} \to m' + n^-)$$
$$mn = (n = 0 \to 0, \mathrm{T} \to m + mn^-)$$

$$\begin{aligned} Th.\ 1.\ m + 0 &= m \\ Proof\ \ m + 0 &= (0 = 0 \to m, \mathrm{T} \to m' + 0^-) \\ &= m. \end{aligned}$$

Only the definition of addition and the properties of conditional expressions were used in this proof.

$Th.$ 2. $(m + n)' = m' + n$

$Proof$ Define $f(m, n) = (n = 0 \to m', \mathrm{T} \to f(m, n^-))$. It is easily seen that $f(m, n)$ converges for all m and n and hence is completely defined by the above equation and is computable from it. Now

$$\begin{aligned} (m + n)' &= (n = 0 \to m, \mathrm{T} \to (m' + n^-)') \\ &= (n = 0 \to m', \mathrm{T} \to (m' + n^-)'), \text{while} \\ m' + n &= (n = 0 \to m', \mathrm{T} \to (m')' + n^-). \end{aligned}$$

It is easily seen that the functions g and h defined by the equations $g(m, n) = (m + n)'$ and $h(m, n) = m' + n$ both satisfy the equation f. For example, it is clear that $g(m', n) = (m' + n^-)'$ and $h(m', n^-) = (m')' + n^-$. Therefore, by the principle of recursion induction h and g are equivalent functions on the domain of where f is defined, but this is the set of all pairs of integers.

The fact that the above defined $f(m, n)$ converges for all m and n is a case of the more general fact that all functions defined by equations of the form

$$\begin{aligned} f(n, x, ..., z) = \ & (n = 0 \to g(x, ..., z), \mathrm{T} \to h(n, x, ..., z, \\ & f(n^-, r(x, ..., z), ..., t(x, ..., z)), \\ & f(n^-, u(x, ..., z), ..., w(x, ..., z)), etc.)) \end{aligned}$$

converge. We are not yet able to discuss formal proofs of convergence.

In presenting further proofs we shall be more terse.

Th. 3. $(m+n)+p = (m+p)+n$.
Proof Let $f(m,n,p) = (p=0 \rightarrow m+n, \mathrm{T} \rightarrow f(m',n,p^-))$. Again f converges for all m,n,p. We have

$$\begin{aligned}
(m+n)+p &= (p=0 \rightarrow m+n, \mathrm{T} \rightarrow (m+n)'+p-) \\
&= (p=0 \rightarrow m+n, \mathrm{T} \rightarrow (m'+n)+p^-) \text{ using Th. 2.} \\
(m+p)+n &= (p=0 \rightarrow m, \mathrm{T} \rightarrow m'+p^-)+n \\
&= (p=0 \rightarrow m+n, \mathrm{T} \rightarrow (m'+p^-)+n).
\end{aligned}$$

Each of these forms satisfies the equation for $f(m,n,p)$.

Setting $m=0$ in Theorem 3 gives

$$(0+n)+p = (0+p)+n$$

so that if we had $0+m=m$ we would have commutativity of addition.

In fact, we cannot prove $0+m=m$ without making some assumptions that take into account that we are dealing with the integers. For suppose our space consisted of the vertices of the binary tree in figure 2, where

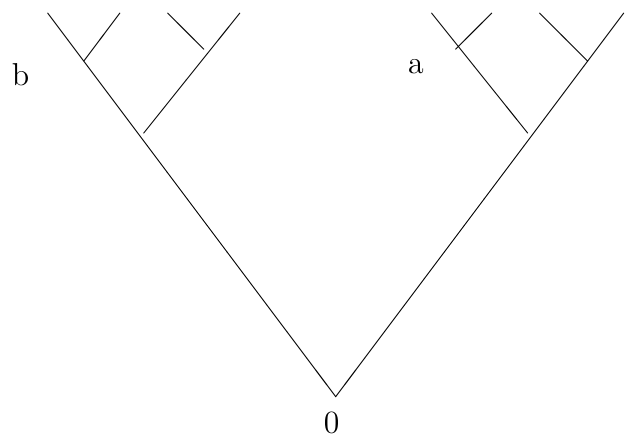

Fig. 2

m' is the vertex just above and to the left, and m^- is the vertex just below, and 0 is the bottom of the tree. $m+n$ can be defined as above and of course satisfies Theorems 1, 2, and 3 but does not satisfy $0 + m = m$. For example, in the diagram $0 + a = b$ although $a + 0 = a$.

We shall make the following assumptions:

1. $m' \neq 0$
2. $(m')^- = m$
3. $(m \neq 0) \supset ((m^-)' = m)$

which embody all of Peano's axioms except the induction axiom.

Th. 4. $0 + n = n$.

$$\begin{aligned} \text{Proof Let } f(n) &= (n = 0 \to 0, \mathrm{T} \to f(n^-)') \\ 0 + n &= (n = 0 \to 0, \mathrm{T} \to 0' + n^-) \\ &= (n = 0 \to 0, \mathrm{T} \to (0 + n^-)') \\ n &= (n = 0 \to n, \mathrm{T} \to n) \\ &= (n = 0 \to 0, \mathrm{T} \to (n^-)') \quad \text{axiom 3} \end{aligned}$$

Th. 5. $m + n = n + m$.

Proof By 3 and 4 as remarked above.

Th. 6. $(m + n) + p = m + (n + p)$

$$\begin{aligned} \text{Proof } (m + n) + p &= (m + p) + n \text{ Th. 3.} \\ &= (p + m) + n \text{ Th. 5.} \\ &= (p + n) + m \text{ Th. 3.} \\ &= m + (n + p) \text{ Th. 5. twice.} \end{aligned}$$

Th. 7. $m \cdot 0 = 0$.

$$\begin{aligned} \text{Proof } m \cdot 0 &= (0 = 0 + 0, T \to m + n \cdot 0^-) \\ &= 0 \end{aligned}$$

Th. 8. $0 \cdot n = 0$.

$$\begin{aligned} \text{Proof Let} f(n) &= (n = 0 \to 0, \mathrm{T} \to f(n^-)) \\ 0 \cdot n &= (n = 0 \to 0, \mathrm{T} \to 0 + 0 \cdot n) = (n = 0 \to 0, \mathrm{T} \to 0 \cdot n) \\ 0 &= (n = 0 \to 0, \mathrm{T} \to 0) \end{aligned}$$

Th. 9. $mn' = m + mn$.

$$\begin{aligned} \text{Proof } mn' &= (n' = 0 \to 0, \mathrm{T} \to m + m \cdot (n')^-) \\ &= m + mn \quad \text{axioms 1 and 2.} \end{aligned}$$

$Th.$ 10. $m(n+p) = mn + mp$.

$$\begin{aligned}
\text{Proof Let} f(m,n,p) &= (p=0 \to mn, \mathrm{T} \to f(m,n',p^-)) \\
m(n+p) &= m(p=0 \to n, \mathrm{T} \to n' + p^-)) \\
&= (p=0 \to mn, \mathrm{T} \to m(n'+p^-)) \\
mn+mp &= mn + (p=0 \to 0, \mathrm{T} \to m + mp^-) \\
&= (p=0 \to mn+0, \mathrm{T} \to mn + (m+mp^-)) \\
&= (p=0 \to mn, \mathrm{T} \to (mn+m)+mp^-) \\
&= (p=0 \to mn, \mathrm{T} \to mn' + mp^-)
\end{aligned}$$

Now we shall give some examples of the application of recursion induction to proving theorems about functions of symbolic expressions. The rest of these proofs depend on an acquaintance with the Lisp formalism.

We start with the basic identities.

$$\begin{aligned}
&car[cons[x;y]] = x \\
&cdr[cons[x;y]] = y \\
&\sim atom[x] \supset cons[car[x];cdr[x] = x \\
&atom[cons[x;y]] = \mathrm{F} \\
&null[x] = eq[x;\mathrm{NIL}]
\end{aligned}$$

Let us define the concatenation x^*y of two lists x and y by the formula

$$x^*y = [null[x] \to y; \mathrm{T} \to cons[car[x]; cdr[x]^*y]]$$

Our first objective is to show that concatenation is associative.

$Th.$ 11. $[x^*y]^*z = x^*[y^*z]$.
Proof

We shall show that $[x^*y]^*z$ and $x^*[y^*z]$ satisfy the functional equation

$$f[x;y;z] = [null[x] \to y^*z; \mathrm{T} \to cons[car[x]; f[cdr[x];y;z]]]$$

First we establish an auxiliary result:

$$cons[a;u]^*v = [null[cons[a;u]] \to v; \mathrm{T} \to cons[car[cons[a;u]]; cdr[cons[a;u]]^*v]] = cons[a;u^*v]$$

Now we write

$$
\begin{aligned}
[x^*y]^*z &= [null[x] \rightarrow y; \mathrm{T} \rightarrow cons[car[x]; cdr[x]^*y]]^*z \\
&= [null[x] \rightarrow y^*z; \mathrm{T} \rightarrow cons[car[x]; cdr[x]^*y]^*z] \\
&= [null[x] \rightarrow y^*z; \mathrm{T} \rightarrow cons[car[x]; cdr[x]^*y]^*z]]
\end{aligned}
$$

and

$$x^*[y^*z] = [null[x] \rightarrow y^*z; \mathrm{T} \rightarrow cons[car[x]; cdr[x]^*[y^*z]]].$$

From these results it is obvious that both $[x^*y]^*z$ and $[x^*[y^*z]$ satisfy the functional equation.

$$
\begin{aligned}
Th.\ 12.\ \mathrm{NIL}^*x &= x \\
x^*\mathrm{NIL} &= x. \\
\text{Proof } \mathrm{NIL}^*x &= [null[\mathrm{NIL}] \rightarrow x; \mathrm{T} \rightarrow cons[car[\mathrm{NIL}]; cdr[\mathrm{NIL}]^*x]] \\
&= x \\
x^*\mathrm{NIL} &= [null[x] \rightarrow \mathrm{NIL}; \mathrm{T} \rightarrow cons[car[x]; cdr[x]^*\mathrm{NIL}]].
\end{aligned}
$$

Let $f[x] = [null[x] \rightarrow \mathrm{NIL}; \mathrm{T} \rightarrow cons[car[x]; f[cdr[x]]]]$. $x^*\mathrm{NIL}$ satisfies this equation. We can also write for any list x

$$
\begin{aligned}
x &= [null[x] \rightarrow x; \mathrm{T} \rightarrow x] \\
&= [null[x] \rightarrow \mathrm{NIL}; \mathrm{T} \rightarrow cons[car[x]; cdr[x]]]
\end{aligned}
$$

which also satisfies the equation.

Next we consider the function $reverse[x]$ defined by

$$reverse[x] = [null[x] \rightarrow \mathrm{NIL}; \mathrm{T} \rightarrow reverse[cdr[x]]^*cons[car[x]; \mathrm{NIL}].$$

It is not difficult to prove by recursion induction that

$$reverse[x^*y] = reverse[y]^*reverse[x]$$

and

$$reverse[reverse[x]] = x.$$

Many other elementary results in the elementary theory of numbers and in the elementary theory of symbolic expressions are provable in the same straightforward way as the above. In number theory one gets as far as the theorem that if a prime p divides ab, then it divides either a or b. However, to formulate the unique factorization theorem requires a notation for dealing with sets of integers. Wilson's theorem, a moderately deep result, can be expressed in this formalism but apparently cannot be proved by recursion induction.

One of the most immediate problems in extending this theory is to develop better techniques for proving that a recursively defined function converges. We hope to find some based on ambiguous functions. However, Godel's theorem disallows any hope that a complete set of such rules can be formed.

The relevance to a theory of computation of this excursion into number theory is that the theory illustrates in a simple form mathematical problems involved in developing rules for proving the equivalence of algorithms. Recursion induction, which was discovered by considering number theoretic problems, turns out to be applicable without change to functions of symbolic expressions.

4 Relation to Other Formalisms

4.1 Recursive function theory

Our characterization of $C\{\mathcal{F}\}$ as the set of functions computable in terms of the base functions in $\mathcal{F}$ cannot be independently verified in general since there is no other concept with which it can be compared. However, it is not hard to show that all partial recursive functions in the sense of Church and Kleene are in $C\{succ, eg\}$. In order to prove this we shall use the definition of partial recursive functions given by Davis [3]. If we modify definition 1.1 of page 41 of Davis [3] to omit reference to oracles we have the following: A function is partial recursive if it can be obtained by a finite number of applications of composition and minimalization beginning with the functions on the following list:

1) x'
2) $U_{i^n}(x_1, ..., x_n) = x_i, \quad 1 \leq i \leq n$
3) $x + y$
4) $x - y = (x - y > 0 \rightarrow x - y, \mathrm{T} \rightarrow 0)$
5) xy

All the above functions are in $C\{succ, eq\}$. Any $C\{\mathcal{F}\}$ is closed under composition so all that remains is to show that $C\{succ, eq\}$ is closed under the minimalization operation. This operation is defined as follows: The operation of minimalization associates with each total function $f(y, x_1, ..., x_n)$ the function $h(x_1, ..., x_n)$ whose value for given $x_1, ..., x_n$ is the least y for which$f(y, x_1, ..., x_n) = 0$, and which is undefined if no such y exists. We have to show that if f is in $C\{succ, eq\}$ so is h. But h may be defined by

$$h(x_1, ..., x_n) = h_2(0, x_1, ..., x_n)$$

where

$$h_2(y, x_1, ..., x_n) = (f(, y, x_1, ..., x_n) = 0 \rightarrow y, \mathrm{T} \rightarrow h_2(y', x_1, ..., x_n)).$$

The converse statement that all functions in $C\{succ, eq\}$ are partial recursive is presumably also true but not quite so easy to prove.

It is our opinion that the recursive function formalism based on conditional expressions presented in this paper is better than the formalisms which have heretofore been used in recursive function theory both for practical and theoretical purposes. First of all, particular functions in which one may be interested are more easily written down and the resulting expressions are briefer and more understandable. This has been observed in the cases we have looked at, and there seems to be a fundamental reason why this is so. This is that both the original Church-Kleene formalism and the formalism using the minimalization operation use integer calculations to control the flow of the calculations. That this can be done is noteworthy, but controlling the flow in this way is less natural than using conditional expressions which control the flow directly.

A similar objection applies to basing the theory of computation on Turing machines. Turing machines are not conceptually different from the automatic computers in general use, but they are very poor in their control structure. Any programmer who has also had to write down Turing machines to compute functions will observe that one has to invent a few artifices and that constructing Turing machines is like programming. Of course, most of the theory of computability deals with questions which are not concerned with the particular ways computations are represented. It is sufficient that computable functions be represented somehow by symbolic expressions, e.g. numbers, and that functions computable in terms of given functions be somehow represented by expressions computable in terms of the expressions representing the original functions. However, a practical theory of computation must be applicable to particular algorithms. The same objection applies to basing a theory of computation on Markov's [9] normal algorithms as applies to basing it on properties of the integers; namely flow of control is described awkwardly.

The first attempt to give a formalism for describing computations that allows computations with entities from arbitrary spaces was made by A. P. Ershov [4]. However, his formalism uses computations with the symbolic expressions representing program steps, and this seems to be an unnecessary complication.

We now discuss the relation between our formalism and computer programming languages. The formalism has been used as the basis for the Lisp programming system for computing with symbolic expressions and has turned out to be quite practical for this kind of calculation. A particular advantage has been that it is easy to write recursive functions that transform programs, and this makes compilers and other program generators easy to write.

The relation between recursive functions and the description of flow control by flow charts is described in Reference 7. An ALGOL program can be described by a recursive function provided we lump all the variables into a single state vector having all the variables as components. If the number of components is large and most of the operations performed involve only a few of them, it is necessary to have separate names for the components. This means that a programming language should include both recursive function definitions and ALGOL-like statements. However, a theory of computation certainly must have techniques for proving algorithms equivalent, and so far it has seemed easier to develop proof techniques like recursion induction for recursive functions than for ALGOL-like programs.

4.2 On the Relations between Computation and Mathematical Logic

In what follows computation and mathematical logic will each be taken in a wide sense. The subject of computation is essentially that of artificial intelligence since the development of computation is in the direction of making machines carry out ever more complex and sophisticated processes, i.e. to behave as intelligently as possible. Mathematical logic is concerned with formal languages, with the representation of information of various mathematical and non-mathematical kinds in formal systems, with relations of logical dependence, and with the process of deduction.

In discussions of relations between logic and computation there has been a tendency to make confused statements, e.g. to say that aspect A of logic is identical with aspect B of computation, when actually there is a relation but not an identity. We shall try to be precise.

There is no single relationship between logic and computation which dominates the others. Here is a list of some of the more important relationships.

1. Morphological parallels

The formal command languages in which procedures are described, e.g. ALGOL; the formal languages of mathematical logic, e.g. first order predicate calculus; and natural languages to some extent: all may be described morphologically (i.e., one can describe what a Grammatical sentence is) using similar syntactical terms. In my opinion, the importance of this relationship has been exaggerated, because as soon as one goes into what the sentences mean the parallelism disappears.

2. Equivalent classes of problems

Certain classes of problems about computations are equivalent to certain classes of problems about formal systems. For example, let

E_1 be the class of Turing machines with initial tapes,
E_2 be the class of formulas of the first order predicate calculus,
E_3 be the class of general recursive functions,
E_4 be the class of formulas in a universal Post canonical system,
E_5 be a class of each element which is a Lisp S-function f together with a suitable set of arguments $a_1, ..., a_k$,
E_6 be a program for a stored program digital computer.

About E_1 we ask: Will the machine ever stop?
About E_2 we ask: Is the formula valid?
About E_3 we ask: Is $f(0)$ defined?
About E_4 we ask: Is the formula a theorem?
About E_5 we ask: Is $f[a_1; ...; a_k]$ defined?
About E_6 we ask: Will the program ever stop?

For any pair (E_i,E_j) we can define a computable map that takes any one of the problems about elements of E_i into a corresponding problem about an element of E_i and which is such that the problems have the same answer. Thus, for any Turing machine and initial tape we can find a corresponding formula of the first order predicate calculus such that the Turing machine will eventually stop if and only if the formula is valid.

In the case of E_6 if we want strict equivalence the computer must be provided with an infinite memory of some kind. Practically, any present computer has so many states, e.g. $2^{36^{15^2}}$, that we cannot reason from finiteness that a computation will terminate or repeat before the solar system comes to an end and one is forced to consider problems concerning actual computers by methods appropriate to machines with an infinite number of states.

These results owe much of their importance to the fact that each of the problem classes is unsolvable in the sense that for each class there is no machine which will solve all the problems in the class. This result can most easily be proved for certain classes (traditionally Turing machine), and then the equivalence permits its extension to other classes. These results have been generalized in various ways. There is the world of Post, Myhill, and others, on creative sets and the work of Kleene on hierarchies of unsolvability. Some of this world is of potential interest for computation even though the generation of new unsolvable classes of problems does not in itself seem to be of great interest for computation.

3. Proof procedures and proof checking procedures

The next relation stems from the fact that computers can be used to carry out the algorithms that are being devised to generate proofs of sentences in various formal systems. These formal systems may have any subject matter of interest in mathematics, in science, or concerning the relation of an intelligent computer program to its environment. The formal system on which the most work has been done is the first order predicate calculus which is particularly important for several reasons. First, many subjects of interest can be axiomatized within this calculus. Second, it is complete, i.e. every valid formula has a proof. Third, although it seems unlikely that the general methods for the first order predicate calculus will be able to produce proofs of significant results in the part of arithmetic axiomatizable in this calculus (or in any other important domain of mathematics), the development of these general methods will provide a measure of what must be left to subject-matter-dependent heuristics. It should be understood by the reader that the first order predicate calculus is undecidable; hence there is no possibility of a program that will decide whether a formula is valid. All that can be done is to construct programs that will decide some cases and will eventually prove any valid formula but which will run on indefinitely in the case of certain invalid formulas.

Proof-checking by computer may be as important as proof generation. It is part of the definition of formal system that proofs be machine checkable. In my forthcoming paper [9], I explore the possibilities and applications of machine checked proofs. Because a machine can be asked to do much more work in checking a proof than can a human, proofs can be made much easier to write in such systems. In particular, proofs can contain a request for the machine to explore a tree of possibilities for a conventional proof. The potential applications for computer-checked proofs are very large. For example, instead of trying out computer programs on test cases until they are debugged, one should prove that they have the desired properties.

Incidentally, it is desirable in this work to use a mildly more general concept of formal system. Namely, a formal system consists of a computable predicate

$$check[statement; proof]$$

of the symbolic expressions *statement* and *Proof*. We say that *Proof* is a proof of *statement* provided

$$check[statement; proof]$$

has the value T.

The usefulness of computer checked proofs depends both on the development of types of formal systems in which proofs are easy to write and on the formalization of interesting subject domains. It should be remembered that the formal systems so far developed by logicians have heretofore quite properly had as their objective that it should be convenient to prove metatheorems about the systems rather than that it be convenient to prove theorems in the systems.

4. Use of formal systems by computer programs

When one instructs a computer to perform a task one uses a sequence of imperative sentences. On the other hand, when one instructs a human being to perform a task one uses mainly declarative sentences describing the situation in which he is to act. A single imperative sentence is then frequently sufficient.

The ability to instruct a person in this way depends on his possession of *common-sense* which we shall define as the fact that we can count on his having available any sufficiently immediate consequence of what we tell him and what we can presume he already knows. In my paper [10] I proposed a computer program called the *Advice Taker* that would have these capabilities and discussed its advantages. The main problem in realizing the *Advice Taker* has been devising suitable formal languages covering the subject matter about which we want the program to think.

This experience and others has led me to the conclusion that mathematical linguists are making a serious mistake in their almost exclusive concentration on the syntax and, even more specially, the grammar of natural languages. It is even more important to develop a mathematical understanding and a formalization of the kinds of information conveyed in natural language.

5 Conclusion: Mathematical Theory of Computation

In the earlier sections of this paper I have tried to lay a basis for a theory of how computations are built up from elementary operations and also of how data spaces are built up. The formalism differs from those heretofore used in the theory of computability in its emphasis on cases of proving statements within the system rather than metatheorems about it. This seems to be a very fruitful field for further work by logicians.

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.

6 REFERENCES

[1] CHURCH, A., *The Calculi of Lambda-Conversion*, Annals of Mathematics Studies, no. 6, Princeton, 1941, Princeton University Press.
[2] –, *Introduction to Mathematical Logic*, Princeton, 1952, Princeton University Press.
[3] DAVIS, M., *Computability and Unsolvability*, New York, 1958, McGraw-Hill.
[4] ERSHOV, A. P., *On Operator Algorithms* (Russian), Doklady Akademii Nauk, vol 122, no. 6, pp. 967-970.
[5] KLEENE, S.C., *Recursive Predicates and Quantifiers*, Transactions of the American Mathematical Society, vol. 53, 1953, p. 41.
[6] MCCARTHY, J., letter to the editor, Communications of the Association for Computing Machinery, vol. 2, August, 1959, p. 2.
[7] –, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, Communications Of the ACM, vol. 3, April, 1960, pp. 184-195.
[8] –, *The LISP Programmer's Manual*, M.I.T. Computation Center, 1960.
[9] –, *Computer Programs for Checking Mathematical Proofs*, to be published in the Proceedings of the American Mathematical Society's Symposium on Recursive Function Theory, held in New York, April, 1961.
[10] –, *Programs With Common Sense*, Proceedings of the Teddington Conference on the Mechanization of Thought Processes, H. M. Stationery Office, 1960.
[11] MARKOV, A.A., *Theory of Algorithms* (Russian), Moscow, 1954, USSR Academy of Sciences, Steklov Mathematical Institute.
[12] NAUR, P., et al., *Report on the Algorithmic Language ALGOL 60*, Communications of the ACM, vol. 3, May 1960.
[13] TURING, A.M., *On Computable Numbers with an Application to the Entscheidungs Problem*, Proceedings of the London Mathematical Society, ser. 2, vol. 43, 1937, p. 230; correction, ibid, vol. 43, 1937, p. 544.
[14] YANOV, Y.I., *The Logical Schemes of Algorithms*, from Problems of Cybernetics I, translated from the Russian by Nadler, Griffiths, Kiss, and Muir, New York, 1960, Pergamon Press Ltd., pp. 82-140.