XIII. ARTIFICIAL INTELLIGENCE*

Prof. J. McCarthy
Prof. M. L. Minsky
Prof. N. Rochester†
Prof. C. E. Shannon
P. W. Abrahams
D. G. Bobrow
R. K. Brayton
L. Hodes
L. Kleinrock
D. C. Luckham
K. Maling
D. M. R. Park
S. R. Russell
J. R. Slagle

A. THE LISP PROGRAMMING SYSTEM

The purpose of this programming system, called LISP (for LISt Processor), is to facilitate programming manipulations of symbolic expressions.

The present status of the system may be summarized as follows:

(a) The source language has been developed and is described in several memoranda from the Artificial Intelligence group.

(b) Twenty useful subroutines have been programmed in LISP, hand-translated into SAP (symbolic machine language for the IBM 704 computer) and checked out on the IBM 704. These include routines for reading and printing list structures.

(c) A routine for differentiating elementary functions has been written. A simple version has been checked out, and a more complicated version that can differentiate any function when given a formula for its gradient is almost checked out.

(d) A universal function apply has been written in LISP, hand-translated, and checked out. Given a symbolic expression for a LISP function and a list of arguments apply computes the result of applying the function to the arguments. It can serve as an interpreter for the system and is being used to check out programs in the LISP language before translating them to machine language.

(e) Work on a compiler has been started. A draft version has been written in LISP, and is being discussed before it is translated to machine language or checked out with apply.

(f) The LISP programming system will be shown in this report to be based mathematically on a way of generating the general recursive functions of symbolic expressions.

The mathematical LISP system is described in more detail in Section XIII-D.

B. ENGINEERING CALCULATIONS IN LISP

The application of the List Processing Language to the calculation of properties of linear passive networks is being studied by N. Rochester, S. Goldberg, C. S. Rubenstein, D. J. Edwards, and P. Markstein. A series of programs in List Processing Language is being written. These will enable the IBM 704 computer to accept a description of a

---

*This work is supported jointly by Research Laboratory of Electronics and the Computation Center, M. I. T.

†Visiting Professor of Electrical Engineering, M. I. T.

network in literal terms and calculate literal expressions for various characteristics of the networks.

The equivalent two-terminal-pair network is among the characteristics to be computed. This will be done by organizing the data into a matrix, inverting it, doing a little more algebra, and then simplifying the result. For a general network of reasonable size, such a result would be too complicated for human consumption. However, if the values of the circuit parameters are all simple multiples of a few values, the result is often simple but the process of reaching it may be too lengthy to be carried out by a person. It is expected that the machine can be made to produce simplified equivalent circuits to describe the actual network with a given degree of accuracy for a given special purpose. The target of this investigation is a general class of computations that have reasonably simple algebraic expressions as answers but require lengthy algebraic manipulation.

C. CHESS

Under the direction of McCarthy and Shannon, Abrahams and Kleinrock have been writing an IBM 704 program for playing chess. The program must keep track of a great deal of information about the state of the chessboard after a contemplated sequence of moves, and rather complicated subprograms are required for this. These subprograms have been checked out and are now being used by Kleinrock and Abrahams.

Abrahams' main effort has gone into writing a two-move checkmate program. This program will read in any chess position from a punched card and find a checkmate for White within two moves if such a mate exists. This is a classic problem in chess puzzles and people often take several hours to solve them. The latest version of our two-move checkmate has solved all problems on which it has been tried within four minutes and will shortly be available as a demonstration program. Our original version considered moves in a virtually random order. Present versions reorder the moves so that moves that have mated or refuted mate in previously examined variations of the same problem are tried first. They also examine certain moves without employing the full machinery of the table-updating subprograms, at a considerable saving of time.

Kleinrock has been working on a series of subprograms to determine whether a particular move is a forcing move. Forcing moves include checks, threats of capture, threats of forks, threats of mate in one, pushing a pawn to the seventh or eighth rank, threats of pins, and several others. Some of these have been checked out; others are in the process of being checked out. These routines are oriented towards combinational play. We hope they will help the program to rediscover many chess "brilliancies" and to find new ones.

Other projects on which work continues include the Advice Taker, visual pattern recognition, and an artificial hand.

(XIII. ARTIFICIAL INTELLIGENCE)

Work has been started by Bobrow, Maling, and Park on a proof checker for predicate calculus, and by Slagle on a program for calculating indefinite integrals.

J. McCarthy, M. L. Minsky, N. Rochester

D. RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS AND THEIR COMPUTATION BY MACHINE

1. Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal (1) for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.

In this report we first describe the class of S-expressions and S-functions. Then we describe the representation of S-functions by S-expressions, which enables us to prove that all computable partial functions have been obtained, to obtain a universal S-function, and to exhibit a set of questions about S-expressions which cannot be decided by an S-function. We describe the representation of S-functions by programs in the IBM 704 computer, and the representation of S-expressions by list structures similar to those used by Newell, Simon, and Shaw (2).

Although we have not carried the development of recursive function theory in terms of S-functions and their representation by S-expressions beyond the simplest theorems, it seems that formulation of this theory in terms of S-functions has important advantages. Devices such as Gödel numbering are unnecessary, and so is the construction of particular Turing machines. (These constructions are all artificial in terms of what is intended to be accomplished by them.) The advantage stems from the fact that functions of symbolic expressions are easily and briefly described as S-expressions, and the representation of S-functions by S-expressions is trivial. Moreover, a large class of S-expression representations of S-functions translate directly into efficient machine

programs for the computation of the functions. Although the functions considered here include all computable functions of S-expressions, describe many inportant processes in a very convenient way, and compile into fast running programs for carrying out the processes, there are other kinds of processes whose description by S-functions is inconvenient and for which the S-functions once found do not naturally compile into efficient programs. For this reason, the LISP system includes the possibility of combining S-functions into Fortran or IAL-like programs. Even this will not provide the flexibility of description of processes hoped for from the Advice Taker system.

2. Functions and Function Definitions

We shall need a number of mathematical ideas and notations concerning functions in general. Most of the ideas are well known but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.

a. Partial Functions

A partial function is a function that is defined only on part of its domain. Partial functions necessarily arise when functions are defined by computations because for some values of the arguments, the computation defining the value of the function may not terminate. However, some of our elementary functions will be defined as partial functions.

b. Propositional Expressions and Predicates

A propositional expression is an expression whose possible values are T (for truth) and F (for falsity). We shall assume that the reader is familiar with the propositional connectives $\wedge$ ("and"), $\vee$ ("or"), and $\sim$ ("not"). Typical propositional expressions are

$x < y$

$(x<y) \wedge (b=c)$

x is prime

A predicate is a function whose range consists of the truth values T and F.

c. Conditional Expressions

The dependence of truth values on the values of quantities of other kinds is expressed in mathematics by predicates, and the dependence of truth values on other truth values by logical connectives. However, the notations for expressing symbolically the dependence of quantities of other kinds on truth values is inadequate, so that English words and phrases are generally used for expressing these dependences in texts that describe other

dependences symbolically. For example, the function $|x|$ is usually defined in words.

Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. A conditional expression has the form

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

where the p's are propositional expressions and the e's are expressions of any kind. It may be read, "If p_1 then e_1, otherwise if p_2 then $e_2, \ldots$, otherwise if p_n then e_n," or "p_1 yields $e_1, \ldots, p_n$ yields e_n."

We now give the rules for determining whether the value of $(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n)$ is defined, and if so what its value is. Examine the p's from left to right. If a p whose value is T is encountered before any p whose value is undefined is encountered, then the value of the conditional expression is the value of the corresponding e (if this is defined). If any undefined p is encountered 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 value of the conditional expression is undefined. We now give examples.

$(1<2 \rightarrow 4, 1 \geq 2 \rightarrow 3) = 4$

$(2<1 \rightarrow 4, 2>1 \rightarrow 3, 2>1 \rightarrow 2) = 3$

$(2<1 \rightarrow 4, T \rightarrow 3) = 3$

$(2<1 \rightarrow \frac{0}{0}, T \rightarrow 3) = 3$

$(2<1 \rightarrow 3, T \rightarrow \frac{0}{0})$ is undefined

$(2<1 \rightarrow 3, 4<1 \rightarrow 4)$ is undefined

Some of the simplest applications of conditional expressions are in giving such definitions as

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

$\delta_{ij} = (i=j \rightarrow 1, T \rightarrow x)$

$\mathrm{sgn}(x) = (x<0 \rightarrow -1, x=0 \rightarrow 0, T \rightarrow 1)$

d. Recursive Function Definitions

By using conditional expressions we can, without circularity, define functions by formulas in which the defined function occurs. For example, we write

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

When we use this formula to evaluate 0! we get the answer 1; because of the way in which the value of a conditional expression was defined, the meaningless expression $0 \cdot (0-1)!$ does not arise. The evaluation of 2! according to this definition proceeds as follows:

$$\begin{aligned} 2! &= (2=0 \rightarrow 1, T \rightarrow 2 \cdot (2-1)!) \\ &= 2 \cdot 1! \\ &= 2 \cdot (1=0 \rightarrow 1, T \rightarrow 1 \cdot (1-1)!) \\ &= 2 \cdot 1 \cdot 0! \\ &= 2 \cdot 1 \cdot (0=0 \rightarrow 1, T \rightarrow 0 \cdot (0-1)!) \end{aligned}$$

$= 2 \cdot 1 \cdot 1$

$= 2$

We now give two other applications of recursive function definitions. The greatest common divisor, gcd(m, n), of two positive integers m and n is computed by means of the Euclidean algorithm. This algorithm is expressed by the recursive function definition:

$$gcd(m, n) = (m > n \rightarrow gcd(n, m), rem(n, m) = 0 \rightarrow m, T \rightarrow gcd(rem(n, m), m))$$

where rem(n, m) denotes the remainder left when n is divided by m.

The Newtonian algorithm for obtaining an approximate square root of a number a, starting with an initial approximation x and requiring that an acceptable approximation y satisfy $|y^2 - a| < \epsilon$, may be written as

$$sqrt(a, x, \epsilon) = (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow sqrt(a, \tfrac{1}{2}(x + \tfrac{a}{x}), \epsilon))$$

The simultaneous recursive definition of several functions is also possible, and we shall use such definitions if they are required.

There is no guarantee that the computation determined by a recursive definition will ever terminate and, for example, an attempt to compute n! from our definition will only succeed if n is a non-negative integer. If the computation does not terminate, the function must be regarded as undefined for the given arguments.

The propositional connectives themselves can be defined by conditional expressions. We write

$$p \wedge q = (p \rightarrow q, T \rightarrow F)$$

$$p \vee q = (p \rightarrow T, T \rightarrow q)$$

$$\sim p = (p \rightarrow F, T \rightarrow T)$$

$$p \supset q = (p \rightarrow q, T \rightarrow T)$$

It is readily seen that the right-hand sides of the equations have the correct truth tables. If we consider situations in which p or q may be undefined, the connectives $\wedge$ and $\vee$ are seen to be noncommutative. For example, if p is false and q is undefined, we see that according to the definitions given above $p \wedge q$ is false, but $q \wedge p$ is undefined. For our applications this noncommutativity is desirable, since $p \wedge q$ is computed by first computing p, and if p is false q is not computed. If the computation for p does not terminate, we never get around to computing q. We shall use propositional connectives in this sense hereafter.

e. Functions and Forms

It is usual in mathematics – outside of mathematical logic – to use the word "function" imprecisely and to apply it to forms such as $y^2 + x$. Because we shall later compute with expressions for functions, we need a distinction between functions and forms and a notation for expressing this distinction. This distinction and a notation for describing it, from which we deviate trivially, is given by Church (3).

Let f be an expression that stands for a function of two integer variables. It should make sense to write f(3, 4) and the value of this expression should be determined. The expression y^2+x does not meet this requirement; y^2+x (3, 4) is not a conventional notation, and if we attempted to define it we would be uncertain whether its value would turn out to be 13 or 19. Church calls an expression like y^2+x a form. A form can be converted into a function if we can determine the correspondence between the variables occurring in the form and the ordered list of arguments of the desired function. This is accomplished by Church's λ-notation.

If $\mathscr{E}$ is a form in variables $x_1, \ldots, x_n$, then $\lambda((x_1, \ldots, x_n), \mathscr{E})$ will be taken to be the function of $\underline{n}$ variables whose value is determined by substituting the arguments for the variables $x_1, \ldots, x_n$ in that order in $\mathscr{E}$ and evaluating the resulting expression. For example, $\lambda((x, y), y^2+x)$ is a function of two variables, and $\lambda((x, y), y^2+x)(3, 4)=13$.

The variables occurring in the list of variables of a λ-expression are dummy or bound, like variables of integration in a definite integral. That is, we may change the names of the bound variables in a function expression without changing the value of the expression, provided that we make the same change for each occurrence of the variable and do not make two variables the same that previously were different. Thus $\lambda((x, y), y^2+x)$, $\lambda((u, v), v^2+u)$ and $\lambda((y, x), x^2+y)$ denote the same function.

We shall frequently use expressions in which some of the variables are bound by λ's and others are not. Such an expression may be regarded as defining a function with parameters. The unbound variables are called free variables.

An adequate notation that distinguishes functions from forms allows an unambiguous treatment of functions of functions. It would involve too much of a digression to give examples here, but we shall use functions with functions as arguments later in this report.

Difficulties arise in combining functions described by λ-expressions, or by any other notation involving variables, because different bound variables may be represented by the same symbol. This is called collision of bound variables. There is a notation involving operators that are called combinators for combining functions without the use of variables. Unfortunately, the combinatory expressions for interesting combinations of functions tend to be lengthy and unreadable.

f. Expressions for Recursive Functions

The λ-notation is inadequate for naming functions defined recursively. For example, using λ's, we can convert the definition

$$\text{sqrt}(a, x, \epsilon)=(|x^2-a|<\epsilon \rightarrow x, T \rightarrow \text{sqrt}(a, \tfrac{1}{2}(x+\tfrac{a}{x}), \epsilon))$$

into

$$\text{sqrt}=\lambda(a, x, \epsilon), (|x^2-a|<\epsilon \rightarrow x, T \rightarrow \text{sqrt}(a, \tfrac{1}{2}(x+\tfrac{a}{x}), \epsilon)))$$

but the right-hand side cannot serve as an expression for the function because there would be nothing to indicate that the reference to sqrt within the expression stood for the expression as a whole.

In order to be able to write expressions for recursive functions, we introduce another notation: label(a, $\mathscr{E}$) denotes the expression $\mathscr{E}$, provided that occurrences of a within $\mathscr{E}$ are to be interpreted as referring to the expression as a whole. Thus we can write

label(sqrt, λ((a, x, ϵ), (|x^2-a|<ϵ→x, T→sqrt(a, $\frac{1}{2}$ (x + $\frac{a}{x}$), ϵ))))

as a name for our sqrt function.

The symbol a in label(a, $\mathscr{E}$) is also bound, that is, it may be altered systematically without changing the meaning of the expression. It behaves differently from a variable bound by a λ, however.

3. Recursive Functions of Symbolic Expressions

We shall first define a class of symbolic expressions in terms of ordered pairs and lists. Then we shall define five elementary functions and predicates, and build from them by composition, conditional expressions, and recursive definitions an extensive class of functions of which we shall give a number of examples. We shall then show how these functions themselves can be expressed as symbolic expressions, and we shall define a universal function apply that allows us to compute from the expression for a given function its value for given arguments. Finally, we shall define some functions with functions as arguments and give some useful examples.

a. A Class of Symbolic Expressions

We shall now define the S-expressions (S stands for symbolic). They are formed by using the special characters

.
)
(

and an infinite set of distinguishable atomic symbols. For atomic symbols, we shall use strings of capital Latin letters and digits with single imbedded blanks. Examples of atomic symbols are

A
ABA
APPLE PIE NUMBER 3

There is a twofold reason for departing from the usual mathematical practice of using single letters for atomic symbols. First, computer programs frequently require hundreds of distinguishable symbols that must be formed from the 47 characters that

are printable by the IBM 704 computer. Second, it is convenient to allow English words and phrases to stand for atomic entities for mnemonic reasons. The symbols are atomic in the sense that any substructure they may have as sequences of characters is ignored. We assume only that different symbols can be distinguished.

S-expressions are then defined as follows:

1. Atomic symbols are S-expressions.
2. If e_1 and e_2 are S-expressions, so is $(e_1 \cdot e_2)$.

Examples of S-expressions are

AB

$(A \cdot B)$

$((AB \cdot C) \cdot D)$

An S-expression is then simply an ordered pair, the terms of which may be atomic symbols or simpler S-expressions. We can represent a list of arbitrary length in terms of S-expressions as follows. The list

$(m_1, m_2, \ldots, m_n)$

is represented by the S-expression

$(m_1 \cdot (m_2 \cdot (\ldots (m_n \cdot NIL) \ldots)))$

Here NIL is an atomic symbol used to terminate lists.

Since many of the symbolic expressions with which we deal are conveniently expressed as lists, we shall introduce a list notation to abbreviate certain S-expressions. We have

1. (m) stands for $(m \cdot NIL)$.
2. $(m_1, \ldots, m_n)$ stands for $(m_1 \cdot (\ldots (m_n \cdot NIL) \ldots))$.
3. $(m_1, \ldots, m_n \cdot x)$ stands for $(m_1 \cdot (\ldots (m_n \cdot x) \ldots))$.

Subexpressions can be similarly abbreviated. Some examples of these abbreviations are

$((AB, C), D)$ for $((AB \cdot (C \cdot NIL)) \cdot (D \cdot NIL))$

$((A, B), C, D \cdot E)$ for $((A \cdot (B \cdot NIL)) \cdot (C \cdot (D \cdot E)))$

Since we regard the expressions with commas as abbreviations for those not involving commas, we shall refer to them all as S-expressions.

b. Functions of S-expressions and the Expressions That Represent Them

We now define a class of functions of S-expressions. The expressions representing these functions are written in a conventional functional notation. However, in order to clearly distinguish the expressions representing functions from S-expressions, we shall use sequences of lower-case letters for function names and variables ranging over the set of S-expressions. We also use brackets and semicolons, instead of parentheses and commas, for denoting the application of functions to their arguments. Thus we write

car[x]

car[cons[(A·B);x]]

In these M-expressions (metaexpressions) any S-expressions that occur stand for themselves.

c. The Elementary S-functions and Predicates

We introduce the following functions and predicates:

1. atom

atom[x] has the value of T or F, accordingly as x is an atomic symbol or not. Thus

atom[X]=T

atom[(X·A)]=F

2. Equality

[x=y] is defined if and only if both x and y are atomic. [x=y]=T if x and y are the same symbol, and [x=y]=F otherwise. Thus

[X=X]=T

[X=A]=F

[X=(X·A)] is undefined.

3. car

car[x] is defined if and only if x is not atomic.

$car[(e_1\cdot e_2)]=e_1$. Thus

car[X] is undefined.

car[(X·A)]=X

car[((X·A)·Y)]=(X·A)

4. cdr

cdr[x] is also defined when x is not atomic. We have

$cdr[(e_1\cdot e_2)]=e_2$. Thus

cdr[X] is undefined.

cdr[(X·A)]=A

cdr[((X·A)·Y]=Y

5. cons

cons[x;y] is defined for any x and y. We have

$cons[e_1;e_2]=(e_1\cdot e_2)$. Thus

cons[X;A]=(X·A)

cons[(X·A);Y]=((X·A)·Y)

car, cdr and cons are easily seen to satisfy the relations

car[cons[x;y]]=x

cdr[cons[x;y]]=y

cons[car[x];cdr[x]]=x, provided that x is not atomic.

The names "car" and "cons" will come to have mnemonic significance only when we discuss the representation of the system in the computer. Compositions of car and cdr give the subexpressions of a given expression in a given position. Compositions of cons form expressions of a given structure out of parts. The class of functions which can be formed in this way is quite limited and not very interesting.

d. Recursive S-functions

We get a much larger class of functions (in fact, all computable functions) when we allow ourselves to form new functions of S-expressions by conditional expressions and recursive definition.

We now give some examples of functions that are definable in this way.

1. ff[x]

The value of ff[x] is the first atomic symbol of the S-expression x with the parentheses ignored. Thus

ff[((A·B)·C)]=A

We have

ff[x]=[atom[x]→x;T→ff[car[x]]]

We now trace in detail the steps in the evaluation of ff[((A·B)·C)]:

ff[((A·B)·C)]=[atom[((A·B)·C)]→((A·B)·C);T→ff[car[((A·B)·C)]]]
=[F→((A·B)·C);T→ff[car[((A·B)·C)]]]
=[T→ff[car[((A·B)·C)]]]
=ff[car[((A·B)·C)]]
=ff[(A·B)]
=[atom[(A·B)]→(A·B);T→ff[car[(A·B)]]]
=[F→(A·B);T→ff[car[(A·B)]]]
=[T→ff[car[(A·B)]]]
=ff[car[(A·B)]]
=ff[A]
=[atom[A]→A;T→ff[car[A]]]
=[T→A;T→ff[car[A]]]
=A

2. subst[x;y;z]

This function gives the result of substituting the S-expression x for all occurrences of the atomic symbol y in the S-expression z. It is defined by

subst[x;y;z]=[atom[z]→[z=y→x;T→z];T→cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]

As an example, we have

subst[(X·A);B;((A·B)·C)]=((A·(X·A))·C)

3. equal[x;y]

This is a predicate that has the value T if x and y are the same S-expression, and

has the value F otherwise. We have

equal[x;y]=[atom[x]∧atom[y]∧[x=y]]∨[~atom[x]∧~atom[y]∧ equal[car[x];car[y]]∧equal[cdr[x];cdr[y]]]

It is convenient to see how the elementary functions look in the abbreviated list notation. The reader will easily verify that

(i) $car[(m_1, m_2, \ldots, m_n)]=m_1$

(ii) $cdr[(m_1, m_2, \ldots, m_n)]=(m_2, \ldots, m_n)$

(iii) cdr[(m)]=NIL

(iv) $cons[m_1;(m_2, \ldots, m_n)]=(m_1, m_2, \ldots, m_n)$

(v) cons[m;NIL]=(m)

We define

null[x]=atom[x]∧[x=NIL]

This predicate is useful in dealing with lists.

Compositions of car and cdr arise so frequently that many expressions can be written more concisely if we abbreviate

cadr[x] for [car[cdr[x]],

caddr[x] for car[cdr[cdr[x]]], etc.

Another useful abbreviation is to write $list[e_1;e_2;\ldots e_n]$ for $cons[e_1;cons[e_2;\ldots;cons[e_n;NIL]\ldots]]$. This function gives the list, $(e_1, \ldots, e_n)$, as a function of its elements.

The following functions are useful when S-expressions are regarded as lists.

1. append[x;y]

append[x;y]=[null[x]→y;T→cons[car[x];append[cdr[x];y]]]

An example is

append[(A, B);(C, D, E)]=(A, B, C, D, E)

2. among[x;y]

This predicate is true if the S-expression x occurs among the elements of the list y. We have

among[x;y]=~null[y]∧[equal[x;car[y]]∨among[x;cdr[y]]]

3. pair[x;y]

This function gives the list of pairs of corresponding elements of the lists x and y. We have

pair[x;y]=[null[x]∧null[y]→NIL;~atom[x]∧~atom[y]→cons[list [car[x];car[y]];pair[cdr[x];cdr[y]]]]

An example is

pair[(A, B, C);(X, (Y, Z), U)]=((A, X), (B, (Y, Z)), (C, U))

4. assoc[x;y]

If y is a list of the form $((u_1, v_1), \ldots, (u_n, v_n))$ and x is one of the u's then assoc[x;y] is the corresponding v. We have

assoc[x;y]=[caar[y]=x→cadar[y];T→assoc[x;cdr[y]]]

An example is

assoc[X;((W, (A, B)), (X, (C;D)), (Y, (E, F)))]=(C, D)

5. sublis[x;y]

Here x is assumed to have the form of a list of pairs $((u_1, v_1), \ldots, (u_n, v_n))$, where the u's are atomic, and y may be any S-expression. The value of sublis[x;y] is the result of substituting each v for the corresponding u in y. In order to define sublis, we first define an auxiliary function. We have

sub2[x;z]=[null[x]→z;caar[x]=z→cadar[x];T→sub2[cdr[x];z]]

and

sublis[x;y]=[atom[y]→sub2[x;y];T→cons[sublis[x;car[y]];sublis[x;cdr[y]]]]

We have

sublis[((X, (A, B)), (Y, (B, C)));(A, X·Y)]=(A, (A, B), B, C)

6. inst[x;y;z]

For this function x is an incomplete list of the form described in connection with sublis. That is, each element of the list x is either of the form (u, v) with u atomic or is of the form (u) with u atomic. The value of inst[x;y;z] is NO if z cannot be obtained as sublis[x*;y], where x* is a completion of x obtained by substituting pairs (u, v) with the same u for the unpaired elements (u). If z can be obtained in this way, inst[x;y;z] is the completed list x*. The purpose of inst is to determine whether z is a substitution instance of the expression y, when substitution is allowed for the variables in x, assuming that we start with all the variables of x unpaired and that all the variables of x occur in y. Rules of inference for logical languages are conveniently written by using inst.

We define an auxiliary function inst2[x;y;z], which is used when y is atomic. We have

inst2[x;y;z]=[null[x]→[equal[y;z]→NIL;
T→NO];caar[x]=y→[null[cdar[x]]→cons[list[caar[x];z];cdr
[x]];equal[cadar[x];z]→x;T→NO];T→inst2[cdr[x];y;z]]

and

inst[x;y;z]=[equal[x;NO]→NO;atom[y]→inst2[x;y;z];atom[z]→NO;T→inst[inst[x;car[y];
car[z]];cdr[y];cdr[z]]]

Here is an example

inst[((X), (Y)), ;(IMPLY, X, Y);(IMPLY, (OR, A, B), (AND, B, C))]=
((X, (OR, A, B,)), (Y, (AND, B, C)))

e. Representation of S-Functions by S-Expressions

S-functions have been described by M-expressions. We now give a rule for translating M-expressions into S-expressions, in order to be able to use S-functions for making certain computations with S-functions and for answering certain questions about S-functions.

The translation is determined by the following rules in which we denote the translation of an M-expression $\mathscr{E}$ by $\mathscr{E}'$.

1. If $\mathscr{E}$ is an S-expression $\mathscr{E}'$ is (QUOTE, $\mathscr{E}$).

2. Variables and function names that were represented by strings of lower-case letters are translated to the corresponding strings of the corresponding upper-case letters. Thus car' is CAR, and subst' is SUBST.

3. A form $f[e_1;\ldots;e_n]$ is translated to $(f', e_1', \ldots, e_n')$. Thus $\{\text{cons}[\text{car}[x];\text{cdr}[x]]\}'$ is (CONS, (CAR, X), (CDR, X)).

4. $\{[p_1 \rightarrow e_1;\ldots;p_n \rightarrow e_n]\}'$ is $(\text{COND}, (p_1', e_1'), \ldots, (p_n', e_n'))$.

5. $\{\lambda[[x_1;\ldots;x_n];\mathscr{E}]\}'$ is $(\text{LAMBDA}, (x_1', \ldots, x_n'), \mathscr{E}')$.

6. $\{\text{label}[a;\mathscr{E}]\}'$ is (LABEL, a', $\mathscr{E}'$).

7. $\{\mathscr{E} = \mathscr{F}\}'$ is (EQ, $\mathscr{E}'$,$\mathscr{F}'$).

With these conventions the substitution function whose M-expression is label[subst; λ[[x;y;z];[atom[z]→[y=z→x;T→z];T→cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]]] has the S-expression

(LABEL, SUBST, (LAMBDA, (X, Y, Z), (COND, ((ATOM, Z), (COND, ((EQ, Y, Z), X), ((QUOTE, T), Z))), ((QUOTE, T), (CONS, (SUBST, X, Y, (CAR, Z)), (SUBST, X, Y, (CDR, Z)))))))

This notation is writable and somewhat readable. It can be made easier to read and write at the cost of making its structure less regular. If more characters were available on the computer, it could be improved considerably.

f. The Universal S-Function apply

There is an S-function apply with the property that if f is an S-expression for an S-function f' and args is a list of arguments of the form (argl, ..., argn), where argl, ...,argn are arbitrary S-expressions, then apply[f;args] and f'[argl;...;argn] are defined for the same values of argl, ..., argn, and are equal when defined. For example,

λ[[x;y];cons[car[x];y]][(A, B);(C, D)]=
apply[(LAMBDA, (X, Y), (CONS, (CAR, X), Y));((A, B), (C, D))]=((A, B), C, D)

The S-function apply is defined by

apply[f;args]=eval[cons[f;appq[args]];NIL]

where

appq[m]=[null[m]→NIL;T→cons[list[QUOTE;car[m]];appq[cdr[m]]]]

and

```
        eval[e;a]=[
atom[e]→eval[assoc[e;a];a];
atom[car[e]]→[
```

```
car[e]=QUOTE→cadr[e];
car[e]=ATOM→atom[eval[cadr[e];a]];
car[e]=EQ→[eval[cadr[e];a]=eval[caddr[e];a]];
car[e]=COND→evcon[cdr[e];a];
car[e]=CAR→car[eval[cadr[e];a]];
car[e]=CDR→cdr[eval[cadr[e];a]];
car[e]=CONS→cons[eval[cadr[e];a];eval[caddr[e];a]];
T→eval[cons[assoc[car[e];a];evlis[cdr[e];a]];a]];
caar[e]=LABEL→eval[cons[caddar[e];cdr[e]];cons[list[cadar[e];car[e];a]];
caar[e]=LAMBDA→eval[caddar[e];append[pair[cadar[e];cdr[e]];a]]]
```

and

```
evcon[c;a]=[eval[caar[c];a]→eval[cadar[c];a];T→evcon[cdr[c];a]]
```

and

```
evlis[m;a]=[null[m]→NIL;T→cons[list[QUOTE;eval[car[m];a]];
evlis[cdr[m];a]]
```

We now explain a number of points about these definitions.

1. apply itself forms an expression representing the value of the function applied to the arguments, and puts the work of evaluating this expression onto a function eval. It uses appq to put quotes around each of the arguments, so that eval will regard them as standing for themselves.

2. eval[e;a] has two arguments, an expression e to be evaluated, and a list of pairs a. The first item of each pair is an atomic symbol, and the second is the expression for which the symbol stands.

3. If the expression to be evaluated is atomic, eval evaluates whatever is paired with it first on the list a.

4. If e is not atomic but car[e] is atomic, then the expression has one of the forms (QUOTE, e) or (ATOM, e) or (EQ, e_1, e_2) or (COND, (p_1, e_1, ..., (p_n, e_n)), or (CAR, e) or (CDR, e) or (CONS, e_1, e_2) or (f, e_1, ..., e_n) where f is an atomic symbol.

In the case (QUOTE;e) the expression e, itself, is taken. In the case of (ATOM, e) or (CAR, e) or (CDR, e) the expression e is evaluated and the appropriate function taken. In the case of (EQ, e_1, e_2) or (CONS, e_1, e_2) two expressions have to be evaluated. In the case of (COND, (p_1, e_1), ..., (p_n, e_n)) the p's have to be evaluated in order until a true p is found, and then the corresponding e must be evaluated. This is accomplished by evcon. Finally, in the case of (f, e_1, ..., e_n) we evaluate the expression that results from replacing f in this expression by whatever it is paired with in the list a.

5. The evaluation of ((LABEL, f, $\mathscr{E}$), e_1, ... e_n) is accomplished by evaluating ($\mathscr{E}$, e_1, ..., e_n) with the pairing (f, (LABEL, f, $\mathscr{E}$) put on the front of the previous list a of pairs.

6. Finally, the evaluation of $((\text{LAMBDA}, (x_1, \ldots x_n), \mathscr{E}), e_1, \ldots, e_n)$ is accomplished by evaluating $\mathscr{E}$ with the list of pairs $((x_1, e_1), \ldots, (x_n, e_n))$ put on the front of the previous list a.

The list a could be eliminated, and LAMBDA and LABEL expressions evaluated by substituting the arguments for the variables in the expressions $\mathscr{E}$. Unfortunately, difficulties involving collisions of bound variables arise, but they are avoided by using the list a.

Calculating the values of functions by using apply is an activity better suited to electronic computers than to people. As an illustration, however, we now give some of the steps for calculating

apply[(LABEL, FF, (LAMBDA, (X), (COND, ((ATOM, X), X), ((QUOTE, T), (FF, (CAR, X))))));((A· B))]=A

The first argument is the S-expression that represents the function ff defined in section 3d. We shall abbreviate it by using the letter ϕ. We have

apply[ϕ;((A· B))]

=eval[(LABEL, FF, ψ), (QUOTE, (A· B)));NIL]

where ψ is the part of ϕ beginning (LAMBDA

=eval[((LAMBDA, (X), ω), (QUOTE, (A·B));((FF, ϕ))]

where ω is the part of ψ beginning (COND

=eval[(COND, (π_1, ϵ), (π_2, ϵ_2));((X, (QUOTE, (A· B))), (FF, ϕ))]

Denoting ((X, (QUOTE, (A· B))), (FF, ϕ)) by α, we obtain

=evcon[((π_1, ϵ_1), (π_2, ϵ_2));α]

This involves evlis[π_1;α]

=eval[(ATOM, X);α]

=atom[eval[X;α]]

=atom[eval[assoc[X;((X, (QUOTE, (A· B))), (FF, ϕ))];α]]

=atom[eval[(QUOTE, (A· B));α]]

=atom[(A· B)]

=F

Our main calculation continues with

apply[ϕ;((A· B))]

=evcon[((π_2, ϵ_2));α],

which involves eval[π_2;α]=eval[QUOTE, T);α]=T.

Our main calculation again continues with

apply[ϕ;((A· B))]

=eval[ϵ_2;α]

=eval[(FF, (CAR, X));α]

=eval[cons[ϕ;evlis[((CAR, X));α]];α]

Evaluating evlis[((CAR, X));α] involves eval[(CAR, X);α]

=car[eval[X;a]]

=car[(A·B)], where we took steps from the earlier computation of atom[eval[X;a]]=A, and so

evlis[((CAR, X;a] then becomes list[list[QUOTE;A]]=((QUOTE, A)), and our main quantity becomes

=eval[(ϕ, (QUOTE, A));a]

The subsequent steps are made as in the beginning of the calculation. The LABEL and LAMBDA cause new pairs to be added to a, which gives a new list of pairs a_1. The π_1 term of the conditional eval[(ATOM, X);a_1] has the value T because X is paired with (QUOTE, A) first in a_1, rather than with (QUOTE, (A·B)) as in a.

Therefore we end up with eval[X;a_1] from the evcon, and this is just A.

g. Functions with Functions as Arguments

There are a number of useful functions some of whose arguments are functions. They are especially useful in defining other functions. One such function is maplist[x;f] with an S-expression argument x and an argument f that is a function from S-expressions to S-expressions. We define

maplist[x;f]=[null[x]→NIL;T→cons[f[x];maplist[cdr[x];f]]]

The usefulness of maplist is illustrated by formulas for the partial derivative with respect to x of expressions involving sums and products of x and other variables. The S-expressions that we shall differentiate are formed as follows.

1. An atomic symbol is an allowed expression.

2. If $e_1, e_2, \ldots, e_n$ are allowed expressions, (PLUS, $e_1, \ldots, e_n$) and (TIMES, $e_1, \ldots, e_n$) are also, and represent the sum and product, respectively, of $e_1, \ldots, e_n$.

This is, essentially, the Polish notation for functions, except that the inclusion of parentheses and commas allows functions of variable numbers of arguments. An example of an allowed expression is (TIMES, X, (PLUS, X, A), Y), the conventional algebraic notation for which is X(X+A)Y.

Our differentiation formula is

diff[y;x]=[atom[y]→[y=x→ONE;T→ZERO];
car[y]=PLUS→cons[PLUS;maplist[cdr[y];λ[[z];diff[
car[z];x]]]];car[y]=TIMES→cons[PLUS;maplist[
cdr[y];λ[[z];cons[TIMES;maplist[cdr[y];λ[[w];z≠w
car[w];T→diff[car[[w];x]]]]]]]]

The derivative of the allowed expression, as computed by this formula, is

(PLUS, (TIMES, ONE, (PLUS, X, A), Y), (TIMES, X, (PLUS, ONE, ZERO), Y,
(TIMES, X, (PLUS, X, A), ZERO))

Besides maplist, another useful function with functional arguments is search, which is defined as

search[x;p;f;u]=[null[x]→u;p[x]→f[x];T→search[cdr[x];p;f;u]

The function search is used to search a list for an element that has the property p, and if such an element is found, f of that element is taken. If there is no such element, the function u of no argument is computed.

4. The LISP Programming System

The LISP programming system is a system for using the IBM 704 computer to compute with symbolic information in the form of S-expressions. In its present state it is being used for the following purposes:

1. Writing a compiler to compile LISP programs into machine language.
2. Writing a program to check proofs in a class of formal logical systems.
3. Writing programs for formal differentiation and integration.
4. Making certain engineering calculations whose results are formulas rather than numbers.
5. Programming the Advice Taker system.

The basis of the system is a way of writing computer programs to evaluate S-functions. This will be described in the following sections.

In addition to the facilities for describing S-functions, there will be facilities for using S-functions in programs written as sequences of statements along the lines of Fortran (4) or IAL (5). These features will not be described in this report.

a. Representation of S-Expressions by List Structure

A list structure is a collection of computer words arranged as in Fig. XIII-1a or 1b. Each word of the list structure is represented by one of the subdivided rectangles in the figure. The left box of a rectangle represents the address field of the word and the right box represents the decrement field. An arrow from a box to another rectangle means that the field corresponding to the box contains the location of the word corresponding to the other rectangle.

It is permitted for a substructure to occur in more than one place in a list structure, as in Fig. XIII-1b, but it is not permitted for a structure to have cycles, as in Fig. XIII-1c.

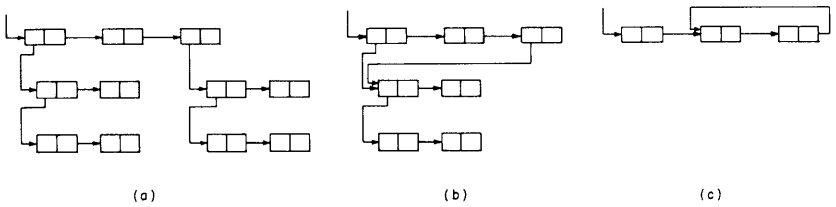

Fig. XIII-1.

An atomic symbol is represented in the computer by a list structure of special form called the association list of the symbol. The address field of the first word contains a special constant which enables the program to tell that this word represents an atomic symbol. We shall describe association lists in section 4b.

An S-expression x that is not atomic is represented by a word, the address and decrement parts of which contain the locations of the subexpressions car[x] and cdr[x], respectively. If we use the symbols A, B, etc. to denote the locations of the association list of these symbols, then the S-expression ((A·B)·(C·(E·F))) is represented by the list structure a of Fig. XIII-2. Turning to the list form of S-expressions, we see that the S-expression (A, (B, C), D), which is an abbreviation for (A· ((B· (C· NIL))· (D· NIL)), is represented by the list structure of Fig. XIII-2b. When a list structure is regarded as representing a list, we see that each term of the list occupies the address part of a word, the decrement part of which points to the word containing the next term, while the last word has NIL in its decrement.

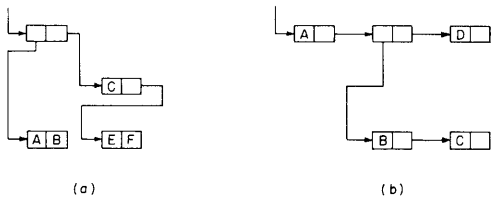

Fig. XIII-2.

An expression that has a given subexpression occurring more than once can be represented in more than one way. Whether the list structure for the subexpression is or is not repeated depends upon the history of the program. Whether or not a subexpression is repeated will make no difference in the results of a program as they appear outside the machine, although it will affect the time and storage requirements. For example, the S-expression ((A·B)·(A·B)) can be represented by either the list structure of Fig. XIII-3a or 3b.

The prohibition against circular list structures is essentially a prohibition against an expression being a subexpression of itself. Such an expression could not exist on paper in a world with our topology. Circular list structures would have some advantages in the machine, for example, for representing recursive functions, but difficulties in printing them, and in certain other operations, make it seem advisable not to use them for the present.

The advantages of list structures for the storage of symbolic expressions are:

1. The size and even the number of expressions with which the program will have to deal cannot be predicted in advance. Therefore, it is difficult to arrange blocks of

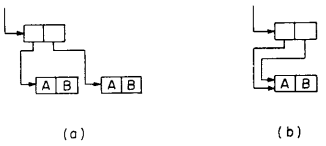

Fig. XIII-3.

storage of fixed length to contain them.

2. Registers can be put back on the free-storage list when they are no longer needed. Even one register returned to the list is of value, but if expressions are stored linearly, it is difficult to make use of blocks of registers of odd sizes that may become available.

3. An expression that occurs as a subexpression of several expressions need be represented in storage only once.

b. Association Lists or Property Lists

In the LISP programming system we put more in the association list of a symbol than is required by the mathematical system described in the previous sections. In fact, any information that we desire to associate with the symbol may be put on the association list. This information may include: the print name, that is, the string of letters and digits which represents the symbol outside the machine; a numerical value if the symbol represents a number; another S-expression if the symbol, in some way, serves as a name for it; or the location of a routine if the symbol represents a function for which there is a machine-language subroutine. All this implies that in the machine system there are more primitive entities than have been described in the sections on the mathematical system.

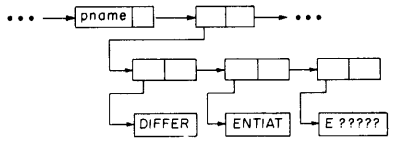

Fig. XIII-4.

For the present, we shall only describe how print names are represented on association lists so that in reading or printing the program can establish a correspondence between information on punched cards, magnetic tape or printed page and the list structure inside the machine. The association list of the symbol DIFFERENTIATE has a segment of the form shown in Fig. XIII-4. Here pname is a symbol that indicates that the structure for the print name of the symbol whose association list this is hangs from the next word on the association list. In the second row of the figure we have a list of three words. The address part of each of these words points to a word containing six 6-bit characters. The last word is filled out with a 6-bit combination that does not represent a character printable by the computer. (Recall that the IBM 704 has a 36-bit word and that printable characters are each represented by 6 bits.) The presence

of the words with character information means that the association lists do not themselves represent S-expressions, and that only some of the functions for dealing with S-expressions make sense within an association list.

c. Free-Storage List

At any given time only a part of the memory reserved for list structures will actually be in use for storing S-expressions. The remaining registers (in our system the number, initially, is approximately 15,000) are arranged in a single list called the free-storage list. A certain register, FREE, in the program contains the location of the first register in this list. When a word is required to form some additional list structure, the first word on the free-storage list is taken and the number in register FREE is changed to become the location of the second word on the free-storage list. No provision need be made for the user to program the return of registers to the free-storage list.

This return takes place automatically, approximately as follows (it is necessary to give a simplified description of this process in this report): There is a fixed set of base registers in the program which contains the locations of list structures that are accessible to the program. Of course, because list structures branch, an arbitrary number of registers may be involved. Each register that is accessible to the program is accessible because it can be reached from one or more of the base registers by a chain of car and cdr operations. When the contents of a base register are changed, it may happen that the register to which the base register formerly pointed cannot be reached by a car-cdr chain from any base register. Such a register may be considered abandoned by the program because its contents can no longer be found by any possible program; hence its contents are no longer of interest, and so we would like to have it back on the free-storage list. This comes about in the following way.

Nothing happens until the program runs out of free storage. When a free register is wanted, and there is none left on the free-storage list, a reclamation cycle starts. First, the program finds all registers accessible from the base registers and makes their signs negative. This is accomplished by starting from each of the base registers and changing the sign of every register that can be reached from it by a car-cdr chain. If the program encounters a register in this process which already has a negative sign, it assumes that this register has already been reached.

After all of the accessible registers have had their signs changed, the program goes through the area of memory reserved for the storage of list structures and puts all the registers whose signs were not changed in the previous step back on the free-storage list, and makes the signs of the accessible registers positive again.

This process, because it is entirely automatic, is more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists. Its

efficiency depends upon not coming close to exhausting the available memory with accessible lists. This is because the reclamation process requires several seconds to execute, and therefore must result in the addition of at least several thousand registers to the free-storage list if the program is not to spend most of its time in reclamation.

d. Elementary S-Functions in the Computer

We shall now describe the computer representations of atom, =, car, cdr, and cons. An S-expression is communicated to the program that represents a function as the location of the word representing it, and the programs give S-expression answers in the same form.

atom. As stated above, a word representing an atomic symbol has a special constant in its address part: atom is programmed as an open subroutine that tests this part. Unless the M-expression atom[e] occurs as a condition in a conditional expression, the symbol T or F is generated as the result of the test. In case of a conditional expression, a conditional transfer is used and the symbol T or F is not generated.

Equality. The program for [e=f] involves testing for the numerical equality of the locations of the words. This works because each atomic symbol has only one association list. As with atom, the result is either a conditional transfer or one of the symbols T or F.

car. Computing car[x] involves getting the contents of the address part of register x. This is essentially accomplished by the single instruction CLA 0, i, where the argument is in index register i, and the result appears in the address part of the accumulator. (We take the view that the places from which a function takes its arguments and into which it puts its results are prescribed in the definition of the function, and it is the responsibility of the programmer or the compiler to insert the required data-moving instructions to get the results of one calculation in position for the next.) ("car" is a mnemonic for "contents of the address part of register.")

cdr. cdr is handled in the same way as car, except that the result appears in the decrement part of the accumulator. ("cdr" stands for "contents of the decrement part of register.")

cons. The value of cons[x;y] must be the location of a register that has x and y in its address and decrement parts, respectively. There may not be such a register in the computer and, even if there were, it would be time-consuming to find it. Actually, what we do is to take the first available register from the free-storage list, put x and y in the address and decrement parts, respectively, and make the value of the function the location of the register taken. ("cons" is an abbreviation for "construct.")

It is the subroutine for cons that initiates the reclamation when the free-storage list is exhausted. In the version of the system that is used at present cons is represented

by a closed subroutine. In the compiled version cons will be open.

e. Representation of S-Functions by Programs

The compilation of functions that are compositions of car, cdr, and cons, either by hand or by a compiler program, is straightforward. Conditional expressions give no trouble except that they must be so compiled that only the p's and e's that are required are computed. However, problems arise in the compilation of recursive functions.

In general (we shall discuss an exception), the routine for a recursive function uses itself as a subroutine. For example, the program for subst[x;y;z] uses itself as a subroutine to evaluate the result into the subexpressions car[z] and cdr[z]. While subst[x;y;cdr[z]] is being evaluated, the result of the previous evaluation of subst[x;y;car[z]] must be saved in a temporary storage register. However, subst may need the same register for evaluating subst[x;y;cdr[z]]. This possible conflict is resolved by the SAVE and UNSAVE routines that use the public push-down list. The SAVE routine is entered at the beginning of the routine for the recursive function with a request to save a given set of consecutive registers. A block of registers called the public push-down list is reserved for this purpose. The SAVE routine has an index that tells it how many registers in the push-down list are already in use. It moves the contents of the registers which are to be saved to the first unused registers in the push-down list, advances the index of the list, and returns to the program from which control came. This program may then freely use these registers for temporary storage. Before the routine exits it uses UNSAVE, which restores the contents of the temporary registers from the push-down list and moves back the index of this list. The result of these conventions is described, in programming terminology, by saying that the recursive subroutine is transparent to the temporary storage registers.

In the hand-compiled version of the system SAVE and UNSAVE are closed subroutines. In the compiler version they will be open.

f. Present Status of the Programming System

At the present time (April 1959), approximately 20 subroutines have been hand-compiled and checked out. These include routines for reading and printing S-expressions and routines for all of the functions that have been mentioned, except those dealing with Turing machines. In particular, apply and its satellites are working. An operator program for apply permits the user to evaluate any S-function with any set of S-expressions as arguments without writing any machine language program: apply then serves as an interpreter for the programming system.

Work has started on a compiler that will produce SAP language routines from the S-expressions for S-functions. Two drafts of the compiler have been written in LISP.

5. Computability and Unsolvability

We shall establish some connections between S-functions and the elementary parts of the theory of computability. Our reason for doing this is to suggest that it may be easier to develop the theory of computable functions in terms of S-functions, rather than in terms of Turing machines or in terms of recursive functions of integer variables.

There are three reasons why S-functions may be appropriate devices for developing the theory of computable functions: First, it is necessary in any form of the theory to establish a correspondence between the functions themselves and a subset of the entities on which the functions operate. Because S-functions operate on symbolic expressions, this correspondence is very natural but in either of the other cases considerable artificiality is involved, especially when Gödel numbers are used. Second, interesting S-functions are computable at a reasonable speed on an electronic computer. Third, the use of conditional expressions simplifies the writing of recursive definitions. This device can be used in any formulation of computability in terms of functions.

The main elementary results of the theory of computable functions, in Turing-machine formulation, are:

1. Turing's thesis – a plausibility argument that any effective computation process can be represented by a Turing machine.

2. The existence of a universal Turing machine that can simulate any other Turing machine.

3. The proof that there is no Turing-machine decision procedure for the problem of whether or not a Turing-machine computation will ever terminate.

The apply function has the same relation to S-functions that a universal Turing machine has to Turing machines.

We might attempt a plausibility argument to the effect that any function that can effectively be computed can be represented by an S-function. Instead, we shall show that there is an S-function that simulates Turing machines.

a. An S-function That Simulates Turing Machines

We now construct an S-function turing[machine;tape], whose arguments are S-expressions representing a Turing machine and an initial tape configuration, and whose value is an S-expression representing the configuration of the tape when the machine stops. (We shall regard a Turing machine as defined by a set of quintuples and assume that it stops when it enters a state and sees a symbol for which there is no corresponding quintuple.) The expression turing[machine;tape] is defined exactly when the simulated calculation terminates.

We proceed as follows:

1. We present a way of describing the instantaneous configurations of a

Turing-machine calculation as S-expressions. Such an S-expression must describe the Turing machine, its internal state, the tape, and the square on the tape that is being read.

2. We give an S-function succ, whose argument is an instantaneous configuration, and whose value is the immediately succeeding configuration, if there is one, and otherwise is 0.

3. We construct from succ the above-mentioned S-function, turing, whose arguments are a Turing machine with a canonical initial state and an initial tape in a standard position, and whose value is defined exactly when the corresponding Turing-machine calculation terminates and which, in that case, is the final tape.

We shall consider Turing machines as given by sets of quintuples. Each quintuple consists of a state, a symbol read, a symbol to be printed, a direction of motion, and a new state. The states are represented by symbols from a given finite set; the symbols, by symbols from another finite set (it does not matter if these sets overlap), and the two directions, by the symbols "L" and "R". A quintuple is then represented by an S-expression (st, sy, nsy, dir, nst). The Turing machine is represented by an S-expression, (ist, blank, quinl, . . . , quink), where ist represents the canonical initial state, and blank is the symbol for a blank square (squares beyond the region explicitly represented in the S-expression for a tape are assumed to be blank and are read that way when they are reached). As an example, we give the representation of a Turing machine which moves to the right on a tape and computes the parity of the number of ones on the tape, ignores zeros, and stops when it comes to a blank square:
(0, B, (0, 0, B, R, 0), (0, 1, B, R, 1), (0, B, 0, R, 2), (1, 0, B, R, 1), (1, 1, B, R, 0), (1, B, 1, R, 2))
It is assumed that the machine will stop if there is no quintuple with a given state-symbol pair, so that this machine stops as soon as it enters state 2.

A Turing-machine tape is represented by an S-expression as follows: The symbols on the squares to the right of the scanned square are given in a list v, the symbols to the left of the scanned square in a list u, and the scanned symbol as a quantity w. These are combined in a list (w, u, v). Thus the tape ...bb11010110b... (with the scanned 0 underlined) is represented by the expression

(0, (1, 0, 1, 1, B, B), (1, 1, 0, B))

We adjoin the state to this triplet to make a quadruplet (s, w, u, v) that describes the instantaneous configuration of a machine.

The function succ[m;c], whose arguments are a Turing machine m and a configuration c, has as its value the configuration that immediately succeeds c; provided that the state-symbol pair scanned in c is listed among the quintuplets of m; otherwise it has the value 0.

The function succ is defined with the aid of auxiliary functions. The first of these, find[st;sy;qs], gives the triplet (nsy;dir;nst) that consists of the last three terms of the

quintuplet of m that contains (st, sy) as its first two elements. The recursive definition is simplified by defining find[st;sy;qs], where qs=cddr[m], since qs then represents the list of quintuplets of m.

```
find[st;sy;qs]=[null[qs]→0;[caar[qs]=st]∧[cadar[qs]=sy]→
caddar[qs];T→find[st;sy;cdr[qs]]]
```

The next auxiliary function is move[m;nsy;tape;dir], which gives the new tape triplet that is obtained by writing nsy on the scanned square of tape and moving in the direction dir.

```
move[m, nsy;tape;dir]=[dir=L→cons[[
null[cadr[tape]]→cadr[m];T→caadr[tape]];
cons[[null[cadr[tape]]→NIL;T→
cdadr[tape]];cons[cons[nsy;
caddr[tape]];NIL]]];dir=R→
cons[[null[caddr[tape]]→
cadr[m];T→caaddr[tape]];
cons[cons[nsy;cadr[tape]];
cons[[null[caddr[tape]]→NIL;T→
cdaddr[tape]];NIL]]]]
```

The reader should not be alarmed at the monstrous size of the last formula. It rises mainly from the operations that take apart and put together the expressions representing the tape and the machine. We have better ways of representing the analysis and synthesis of expressions of fixed form, but they will not be described in this report.

We now have

```
succ[m;c]=[find[car[c];cadr[c];cddr[m]]
=0→0;T→cons[caddr[find[
car[c];cadr[c];cddr[m]]];
cddr[m]]];cdr[c];cadr[find[car[e];cadr[c];cddr[m]]]]]]
```

Finally, we define

```
turing[m;tape]=tu[m;cons[car[m];tape]]
```

where

```
tu[m;c]=[succ[m;c]=0→car[c];1→tu[m;succ[m;c]]]
```

b. A Question about S-Functions Which is Undecidable by an S-Function

It was mentioned in section 2 that a recursive function is not defined for values of its arguments for which the process of evaluation does not terminate. It is easy to give examples of S-functions that are defined for all arguments, examples that are defined for no arguments, and examples that are defined for some arguments. It would be nice to be able to determine whether a given S-Function is defined for given arguments. Consider, then the predicate def[f;s], where f is the

S-expression for an S-function ϕ, and s is a list of arguments, and whose value is T if $\phi[s]$ is defined, and is F otherwise.

We assert that def[f;s] is not an S-function. (If we assume Turing-machine theory, this is an obvious consequence of the results of the last section but, in support of the contention that S-functions are a good vehicle for expounding the theory of computability, we give a separate proof.)

THEOREM. def[f;s] is not an S-function.

PROOF. Suppose the contrary. Consider the function

$$g=\lambda[[f];[\sim def[f;f]\rightarrow T, T\rightarrow car[NIL]]]$$

If def were an S-function, g would also be an S-function. For any S-function u with S-expression u', g[u'] is T if u[u'] is undefined, and is undefined otherwise.

Consider now g[g'], where g' is an S-expression for g. Assume, on the one hand, that g[g'] is defined. This is precisely the condition that g' be the kind of S-expression for which g is undefined. On the other hand, were g[g'] undefined, g' would be the kind of S-expression for which g is defined.

Thus our assumption that def[f;s] is an S-function leads to a contradiction.

This proof is similar to the corresponding proof in Turing-machine theory. The simplicity of the rules by which S-functions are represented as S-expressions makes the development from scratch simpler, however.

c. Recursive Functions of Integers

The computational universality of S-functions is dependent more on the use of conditional expressions and recursion than on the base functions and predicates used. We can, for example, base the recursive functions of non-negative integers on the successor function (the successor of n is denoted by n') and the predicate equality. We define

$$dim[m]=dim[m;0]$$

where

$$dim[m;n]=[m=0\rightarrow 0;n'=m\rightarrow n;T\rightarrow dim[m;n']]$$

We further define

$$m+n=[n=0\rightarrow m;T\rightarrow m'+dim[n]]$$

and

$$m*n=[n=0\rightarrow 0, T\rightarrow m+n*dim[n]]$$

The general recursive functions of integers can be built up by continuing this process, but special devices would still be required if the theory were to be self-applied.

6. A Variant of LISP

There are a number of ways of defining functions of symbolic expressions which are quite similar to the system we have adopted. Each of them involves three basic functions, conditional expressions, and recursive function definitions, but the class of

expressions corresponding to S-expressions is different, and so are the precise definitions of the functions. We shall describe one of these variants called linear LISP.

The L-expressions are defined as follows:

1. A finite list of characters is admitted.
2. Any string of admitted characters is an L-expression.

This includes the null string denoted by Λ.

There are three functions of strings:

1. first[x] is the first character of the string x.

   first[Λ] is undefined.

For example, first [ABC]=A.

2. rest[x] is the string of characters which remains when the first character of the string is deleted.

   rest[Λ] is undefined.

For example, rest[ABC]=BC.

3. combine[x;y] is the string formed by prefixing the character x to the string y.

For example, combine[A;BC]=ABC.

There are three predicates on strings:

1. char[x], x is a single character.
2. null[x], x is the null string.
3. x=y, defined for x and y characters.

The advantage of linear LISP is that no characters are given special roles, as are parentheses, dots, and commas in LISP. This permits computations with all expressions that can be written linearly. The disadvantage of linear LISP is that the extraction of subexpressions is a fairly involved, rather than an elementary, operation. It is not hard to write, in linear LISP, functions that correspond to the basic functions of LISP, so that, mathematically, linear LISP includes LISP. This turns out to be the most convenient way of programming, in linear LISP, the more complicated manipulations. However, if the functions are to be represented by computer routines, LISP is essentially faster.

7. Flow Charts and Recursion

Since both the usual form of computer program and recursive function definitions are universal computationally, it is interesting to display the relation between them. The translation of recursive symbolic functions into computer programs will be the subject of most of the rest of this report. In this section we show how to go the other way, at least in principle.

The state of the machine at any time during a computation is given by the values of a number of variables. Let these variables be combined into a vector ξ. Consider a program block with one entrance and one exit. It defines and is essentially defined by

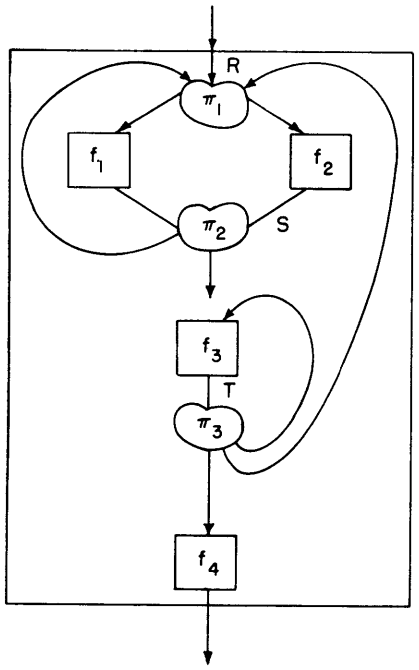

Fig. XIII-5.

a certain function f that takes one machine configuration into another, that is, f has the form $\xi'=f(\xi)$. Let us call f the associated function of the program block. Now let a number of such blocks be combined into a program by decision elements π that decide after each block is completed which block will be entered next. Nevertheless, let the whole program still have one entrance and one exit.

We give as an example the flow chart of Fig. XIII-5. Let us describe the function $r[\xi]$ that gives the transformation of the vector ξ between entrance and exit of the whole block. We shall define it in conjunction with the functions $s[\xi]$ and $t[\xi]$, which give the transformations that ξ undergoes between the points S and T, respectively, and the exit. We have

$$r[\xi]=[\pi_{11}[\xi]\rightarrow s[f_1[\xi]];T\rightarrow s[f_2[\xi]]]$$
$$s[\xi]=[\pi_{21}[\xi]\rightarrow r[\xi];T\rightarrow t[f_3[\xi]]]$$
$$t[\xi]=[\pi_{31}[\xi]\rightarrow f_4[\xi];\pi_{32}[\xi]\rightarrow r[\xi];T\rightarrow t[f_3[\xi]]]$$

Given a flow chart with a single entrance and a single exit, it is easy to write down the recursive function that gives the transformation of the state vector from entrance to exit in terms of the corresponding functions for the computation blocks and the predicates of the branch points. In general, we proceed as follows.

In Fig. XIII-6 let β be an n-way branch point, and let $f_1, \ldots, f_n$ be the computations leading to branch points $\beta_1, \beta_2, \ldots, \beta_n$. Let ϕ be the function that transforms ξ between β and the exit of the chart, and let $\phi_1, \ldots, \phi_n$ be the corresponding functions for $\beta_1, \ldots, \beta_n$. We then write

$$\phi[\xi]=[p_1[\xi]\rightarrow\phi_1[f[\xi]];\ldots;p_n[\xi]\rightarrow\phi_n[f_n[\xi]]]$$

There are, however, problems connected with the descriptions of the functions f that describe the computation blocks. They are

1. Each block usually affects only a few of the components of the vector ξ and is best described in terms that involve only the components affected and those on which they depend. Such a description is not affected by adding other components to the total vector ξ for other parts of the program.

The so-called arithmetic, or replacement, statement of Fortran, IT, or IAL, which has the form $A=\mathscr{E}\{B, C\ldots\}$, is of this kind. Even input and output statements can be included in this scheme, provided that the states of external media are included in the vector ξ.

2. A sub-block in a flow chart may also be recursively defined. Usually, the

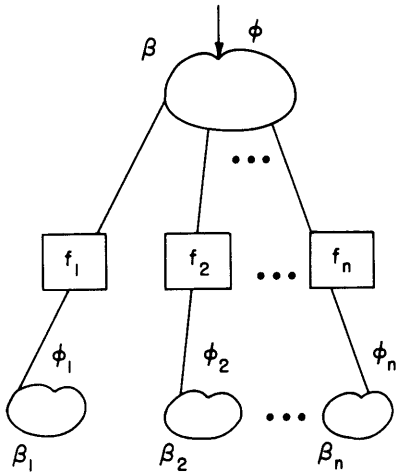

Fig. XIII-6.

vector for the sub-block will be conveniently described as having a different set of components from the program as a whole. There will be variables (i.e., components) internal to the sub-block, and variables that are not involved in the subcalculation. To adequately describe this, a notation for extensions and reductions of vector functions is required.

3. Even with a notation for extensions and reductions, subroutines require additional treatments. A subroutine occurring at several places in the program will not usually be applied to the same components at each place. In order to use the same subroutine at different places, it seems necessary to have a notation for selection and permutation of variables.

Acknowledgments

The inadequacy of the λ-notation for naming recursive functions was noticed by N. Rochester, and he discovered an alternative to the solution involving label which has been used here. The form of subroutine for cons which permits its composition with other functions was invented, in connection with another programming system, by C. Gerberick and H. L. Gelernter, of IBM Corporation. An earlier version of an S-function for simulating a Turing machine was written by P. W. Abrahams.

J. McCarthy

References

1. J. McCarthy, Programs with common sense, paper presented at the Symposium on the Mechanization of Thought Processes, National Physical Laboratory, Teddington, England, Nov. 24-27, 1958. (Published in Proceedings of the Symposium).

2. A. Newell and J. C. Shaw, Programming the logic theory machine, Proc. Western Joint Computer Conference, Feb. 1957.

3. A. Church, The Calculi of λ-conversion (Princeton University Press, Princeton, N. J., 1941).

4. FORTRAN Programmer's Reference Manual, IBM Corporation, New York, Oct. 15, 1956.

5. A. J. Perlis and K. Samelson, International algebraic language, Preliminary Report, Communications of the Association for Computing Machinery, Dec. 1958.