# The mechanical evaluation of expressions

*By* P. J. Landin

**This paper is a contribution to the "theory" of the activity of using computers. It shows how some forms of expression used in current programming languages can be modelled in Church's λ-notation, and then describes a way of "interpreting" such expressions. This suggests a method, of analyzing the things computer users write, that applies to many different problem orientations and to different phases of the activity of using a computer. Also a technique is introduced by which the various composite information structures involved can be formally characterized in their essentials, without commitment to specific written or other representations.**

## Introduction

The point of departure of this paper is the idea of a machine for evaluating schoolroom sums, such as

1. $(3 + 4)(5 + 6)(7 + 8)$
2. **if** $2^{19} < 3^{12}$ **then** $\sqrt[12]{2}$ **else** $\sqrt[53]{2}$
3. $\sqrt{\left(\dfrac{17 \cos \pi/17 - \sqrt{(1 - 17 \sin \pi/17)}}{17 \cos \pi/17 + \sqrt{(1 + 17 \sin \pi/17)}}\right)}$

Any experienced computer user knows that his activity scarcely resembles giving a machine a numerical expression and waiting for the answer. He is involved with flow diagrams, with replacement and sequencing, with programs, data and jobs, and with input and output. There are good reasons why current information-processing systems are ill-adapted to doing sums. Nevertheless, the questions arise: Is there any way of extending the notion of "sums" so as to serve some of the needs of computer users without all the elaborations of using computers? Are there features of "sums" that correspond to such characteristically computerish concepts as flow diagrams, jobs, output, etc.?

This paper is an introduction to a current attempt to provide affirmative answers to these questions. It leaves many gaps, gets rather cursory towards the end and, even so, does not take the development very far. It is hoped that further piecemeal reports, putting right these defects, will appear elsewhere.

## Expressions

### Applicative structure

Many symbolic expressions can be characterized by their "operator/operand" structure. For instance

$$a/(2b + 3)$$

can be characterized as the expression whose operator is '/' and whose two operands are respectively '$a$,' and the expression whose operator is '+' and whose two operands are respectively the expression whose operator is '×' and whose two operands are respectively '2' and '$b$,' and '3.' Operator/operand structure, or "applicative" structure, as it will be called here, can be exhibited more clearly by using a notation in which each operator is written explicitly and prefixed to its operand(s), and each operand (or operand-list) is enclosed in brackets, e.g.

$$/(a, +(\times(2, b), 3)).$$

This notation is a sort of standard notation in which all the expressions in this paper could (with some loss of legibility) be rendered.

The following remarks about applicative structure will be illustrated by examples in which an expression is written in two ways: on the left in some notation whose applicative structure is being discussed, and on the right in a form that displays the applicative structure more explicitly, e.g.

| | |
|---|---|
| $a/(2b + 3)$ | $/(a, +(\times(2, b), 3))$ |
| $(a + 3)(b - 4) +$ | $+(\times(+(a, 3), -(b, 4)),$ |
| $(c - 5)(d - 6)$ | $\times(+(c, 5), -(d, 6)))$. |

In both these examples the right-hand version is in the "standard" notation. In most of the illustrations that follow, the right-hand version will not adhere rigorously to the standard notation. The particular point illustrated by each example will be more clearly emphasized if irrelevant features of the left-hand version are carried over in non-standard form. Thus the applicative structure of subscripts is illustrated by

| | |
|---|---|
| $a_j b_{jk}$ | $a(j)b(j, k)$. |

Some familiar expressions have features that offer several alternative applicative structures, with no obvious criterion by which to choose between them. For example

$$3 + 4 + 5 + 6 \qquad \begin{cases} +(+(+(3, 4), 5), 6) \\ +(3, +(4, +(5, 6))) \\ \Sigma'(3, 4, 5, 6) \end{cases}$$

where $\Sigma'$ is taken to be a function that operates on a list of numbers and produces their sum. Again

$$a^2 \qquad \begin{cases} \uparrow(a, 2) \\ \textit{square}\ (a) \end{cases}$$

where $\uparrow$ is taken to be exponentiation.

Sometimes the choice may be more material to the meaning. For instance, without background information it is impossible to decide whether or not

$$f(y + 1) + n(y - 1)$$

contains a sub-expression whose operator is multiplication. We are not concerned here with offering specific rules for answering such questions. What interests us is that in many cases such a rule can be considered as a rule about applicative structure.

Using Church's $\lambda$-notation [1] we can impute applicative structure to some familiar notations that use "private" (or "internal," or "local," or "dummy," or "bound") variables, such as the second and third occurrence of '$x$' in the following:

$$\int_0^x x^2 dx \qquad \int(0, x, \lambda x.x^2).$$

Similarly

$$\Sigma_{0 \leqslant j < n} a_{ij} b_{jk} \qquad \Sigma''(0, n, \lambda j.a(i, j)b(j, k))$$

where $\Sigma''$ is a triadic function that is analogous to $\int$.

**Auxiliary definitions**

The use of auxiliary definitions to qualify an expression can also be rendered in terms of $\lambda$. E.g.

$$(u - 1)(u + 2) \qquad \{\lambda u.(u - 1)(u + 2)\}[7 - 3]$$
$$\textbf{where } u = 7 - 3.$$

Notice that $\lambda u.(u - 1)(u + 2)$ is a function and hence it is appropriate to write this expression in a context that is more familiarly occupied by an identifier, such as '*sin*' or '*f*,' designating a function. Notice also that an expression that denotes a function does not necessarily occur in such a context; witness some previous examples and also

$$\int (0, \pi/2, sin).$$

We shall consistently distinguish between "operators" and "functions" as follows. An *operator* is a sub-expression of a (larger) expression appearing in a context that, when written in standard form, would have an operand (or operand-list) to the right of it. A *function* bears the same relation to an operator as a number, e.g. the fourth non-negative integer, does to a numerical expression, e.g. $(16 - 7)/(5 - 2)$. The "value" of this expression is a number; similarly we shall speak of the "value" of an expression that can occur as an operator. Just as the value of an expression that occurs as an operand combined with '$\sqrt{}$' must, to make sense, be a number, so the value of an expression that occurs as an operator must be a function. However, any expression that can occur sensibly as an operator can also occur sensibly as an operand.

Applicative structure can be indicated unambiguously by brackets. Legibility is improved by using a variety of bracket shapes. In particular we shall tend to use braces for enclosing (long) operators and square brackets for enclosing operands (and operand-lists). However, except that we observe correct mating, no formal significance will be attached to differences of bracket shape. That is to say, the rules for making sense of a written expression do not rely on them; no information would be lost by disregarding the differences in a correctly mated expression.

There is another informal device by which we shall bring out the internal grouping of long expressions, namely indentation. For instance, the connection between the items of an operand-list, or the two components of an operator/operand combination, will frequently be emphasized by indenting them equally.

The use of several auxiliary definitions, rather than just one, can be rendered in terms of $\lambda$. For example, if the definitions are mutually independent, they can be considered as a "simultaneous," or "parallel" definition of several identifiers, e.g.

$$u = 2p + q \qquad (u, v) = (2p + q, p - 2q).$$
$$\textbf{and } v = p - 2q$$

So if Church's notation is extended to permit a *list* of identifiers between the '$\lambda$' and the '.', a group of mutually independent auxiliary definitions raises no new issue, e.g.

$$u(u + 1) - v(v + 1) \qquad \{\lambda(u, v).u(u + 1) - v(v + 1)\}$$
$$\textbf{where } u = 2p + q \qquad [2p + q, p - 2q]$$
$$\textbf{and } v = p - 2q.$$

If the definitions are inter-dependent the correspondence is more elaborate. Some examples of this situation will be given below.

When we say that the applicative structure of a specific piece of algebraic notation is such-and-such, we are providing unique answers to certain questions about it, such as "What is its operator?" "What are its operands?" Our discussion of *specific* algebraic notations will now be interrupted by a discussion of what precisely these questions are. That is to say, the next Section is devoted to explaining what is meant by "applicative structure" rather than to exhibiting the applicative structure of specific notations.

This attempt to characterize applicative structure will use a particular technique called here "structure definitions," and used later in the paper to characterize other sorts of structure. The next Section but one explains this technique. After these two Sections, the discussion of the applicative structure of *specific* notations will be resumed.

**Applicative expressions**

The expressions in this paper are constructed out of certain basic components which are, for our purposes, "atomic"; i.e. their internal structure (if any) does not concern us. They comprise single- and multi-character constants and variables, including decimal numbers. All these will be called *identifiers*. There will be no need for a more precise characterization of identifiers.

By a *λ-expression* we mean, provisionally, an expression characterized by two parts: its *bound variable* part, written between the 'λ' and the '.'; and its *λ-body*, written after the '.'. (A more precise characterization appears below.)

Some of the right-hand versions appearing above contain a λ-expression. Some of those below contain several λ-expressions, sometimes one inside another. This paper shows that many expressions can be considered as constructed out of identifiers in three ways: by forming λ-expressions, by forming operator/operand combinations, and by forming lists of expressions. Of these three ways of constructing composite expressions, the first two are called "functional abstraction" and "functional application," respectively. We shall show below that the third way can be considered as a special case of functional application and so, in so far as our discussion refers to functional application, it implicitly refers also to this special case.

We are, therefore, interested in a class of expressions about any one of which it is appropriate to ask the following questions:

Q1. Is it an identifier? If so, what identifier?

Q2. Is it a λ-expression? If so, what identifier or identifiers constitute its bound variable part and in what arrangement? Also what is the expression constituting its λ-body?

Q3. Is it an operator/operand combination? If so, what is the expression constituting its operator? Also what is the expression constituting its operand?

We call these expressions *applicative expressions* (AEs).

Later the notion of the "value of" (or "meaning of," or "unique thing denoted by") an AE will be given a formal definition that is consistent with our correspondence between AEs and less formal notations. We shall find that, roughly speaking, an AE denotes something as long as we know the value of each identifier that occurs free in it, and provided also that the expression does not associate any argument with a function that is not applicable to it. In particular, for a combination to denote something, its operator must denote a function that is applicable to the value of its operand. On the other hand, any λ-expression denotes a function; roughly speaking, its domain (which might have few members, or even none) contains anything that makes sense when substituted for occurrences of its bound variable throughout its body.

Given a mathematical notation it is a trivial matter to find a correspondence between it and AEs. It is less trivial to discover one in which the intuitive meaning of the notation corresponds to the value of AEs in the sense just given. A correspondence that meets this condition might be called a "semantically acceptable" correspondence. For instance, someone might conceivably denote the sum

$$v_r + v_{r+1} + \ldots + v_{r+s-1}$$

of a segment of a vector by

$$v_r^{(s)}.$$

The most direct rendering of this as an AE is something like

$$sum\ (v(r), s).$$

However, this is not a semantically acceptable correspondence since it wrongly implies dependence on only one element of $v$, namely $v_r$. The same criterion prevents λ from being considered as an operator, in our sense of that word; more precisely it rules that

$$\lambda(x, x^2 + 1)$$

incorrectly exhibits the applicative structure of '$\lambda x . x^2 + 1$.'

We are interested in finding semantically acceptable correspondences that enable a large piece of mathematical symbolism (with supporting narrative) to be rendered by a single AE.

**Structure definitions**

AEs are a particular sort of composite information structure. Lists are another sort of composite information structure. Several others will be used below, and they will be explained in a fairly uniform way, each sort being characterized by a "structure definition." A *structure definition* specifies a class of composite information structures, or *constructed objects* (COs) as they will be called in future. It does this by indicating how many components each member of the class has and what sort of object is appropriate in each position; or, if there are several alternative formats, it gives this information in the case of each alternative. A structure definition also specifies the identifiers that will be used to designate various operations on members of the class, namely some or all of the following:

(*a*) *predicates* — for testing which of the various alternative formats (if there are alternatives) is possessed by a given CO;

(*b*) *selectors* — for selecting the various components of a given CO once its format is known;

(*c*) *constructors* — for constructing a CO of given format from given components.

The questions Q1 to Q3 above comprise the main part of the structure definition for AEs. What they do not convey is the particular identifiers to be used to designate the predicates, selectors and constructors. Future structure definitions in this paper will be laid out in roughly the following way:

An AE is either

an *identifier*,

or a *λ-expression* (*λexp*) and has a *bound variable* (*by*) which is an identifier or identifier-list,
and a *λ-body* (*body*) which is an AE,

or a *combination* and has an *operator* (*rator*)
which is an AE,
and an *operand* (*rand*)
which is an AE.

This is intended to indicate that '*identifier*,' '*λ-expression*' and '*combination*' (and also the abbreviations written after them if any) designate the predicates, and '*bv*,' '*body*,' '*rator*,' '*rand*' (mentioning here the abbreviated forms) designate the selectors. We consider a predicate to be a function whose result for any suitable argument is a "truth-value," i.e. either **true** or **false**. For instance, if $X$ is a λ-expression, then the predicate *λexp* applied to $X$ yields **true**, whereas *identifier* yields **false**; i.e. the following equations hold:

$$\lambda exp\ X = \textbf{true}$$
$$identifier\ X = \textbf{false}.$$

(It will be observed that, by considering predicates as functions, we are led into a slight conflict with the normal use of the word "apply." For instance, in normal use it might be said that the predicate *even* "applies to" the number six, and "does not apply to" the number seven. We must here avoid this turn of phrase and say instead that *even* "holds for," or "yields **true** when applied to," six; and "does not hold for," or "yields **false** when applied to," seven.)

The constructors will not usually be named explicitly. Instead we shall use obviously suggestive identifiers such as '*constructλexp*.' E.g. the following equations hold:

$$\begin{aligned} &identifier\ (construct\lambda exp\ (J, X)) &&= \textbf{false} \\ &\lambda exp\ (construct\lambda exp\ (J, X)) &&= \textbf{true} \\ &bv\ (construct\lambda exp\ (J, X)) &&= J \\ &construct\lambda exp\ (bv\ X, body\ X) &&= X \end{aligned}$$

and many others. (More precisely, each of these equations holds provided $J$ and $X$ are such as to make both sides meaningful. Thus the first three hold provided $J$ is an identifier or list of identifiers and $X$ is an AE. Again, the last holds provided $X$ is a λ-expression.)

A structure definition can also be written more formally, as a definition with a left-hand side and a right-hand side. The left-hand side consists of all the identifiers to which the structure definition gives meaning. The right-hand side is an AE containing references to the component-classes involved (e.g. some class of character-strings, in the case of AEs that are identifiers) and also to one or more of a small number of functions concerned with classes of COs. However, in this paper we shall not formalize the notion of structure definitions, and shall write any we need in the style illustrated above.

**Function definitions**

In ordinary use, definitions frequently give a functional, rather than numerical, meaning to the definiendum by using a dummy argument variable. This can be rendered as an explicit definition with a λ-expression for its right-hand side, e.g.

$$f(y) = y(y+1) \qquad f = \lambda y.y(y+1).$$

So an expression using an auxiliary function definition can be rendered by using two λ-expressions, one for its operator and one for its operand, e.g.

| | |
|---|---|
| $f(3) + f(4)$ | $\{\lambda f.f(3) + f(4)\}[\lambda y.y(y+1)]$. |
| **where** $f(y) = y(y+1)$ | |

A group of auxiliary definitions may include both numerical and functional definitions, e.g.

| | |
|---|---|
| $f(a+b, a-b) + f(a-b, a+b)$ | $\{\lambda(a, b, f).f(a+b, a-b) + f(a-b, a+b)\}$ |
| **where** $a = 33$ | $[33, 44, \lambda(u, v).uv(u+v)]$. |
| **and** $b = 44$ | |
| **and** $f(u, v) = uv(u+v)$ | |

When a λ-expression is written as a sub-expression of a larger expression, the question may arise: how far to the right does its body extend? This question can always be evaded by using enough brackets, e.g.

$$(\lambda(u, v).(uv(u+v))).$$

However, to economize in brackets, we adopt the convention that it extends as far as is compatible with the brackets, except that it is stopped by a comma. Another way of saying this is that the "binding power" of the '.' is less than that of functional application, multiplication and all the written operators such as '+,' '/,' etc., but exceeds that of the comma. For example:

| | |
|---|---|
| $f(g(a)) + g(f(b))$ | $\{\lambda(f, g).f(g(a)) + g(f(b))\}$ |
| **where** $f(z) = z^2 + 1$ | $[\lambda z.z^2 + 1, \lambda z.z^2 - 1]$. |
| **and** $g(z) = z^2 - 1$ | |

An identifier may occur in the *bound variable* part of a λ-expression (either constituting the entire bound variable, or as one item of it). Apart from this, every written occurrence of AE is in one of the following four sorts of context:

(*a*) It is the λ-body of some λ-expression.
(*b*) It is the operator of some combination.
(*c*) It is the operand of some combination.
(*d*) It is a "complete" AE occurring in a context of English narrative, or other non-AE.

Each of the three formats of AE can appropriately appear in any of the four sorts of contexts. We have already seen that λ-expressions, like identifiers, can appropriately occur both as operators and as operands. Below we shall find combinations appearing as operators, and λ-expressions appearing as λ-bodies. These last two possibilities are both associated with the possibility that a function might produce a function as its result. Together with more obviously acceptable possibilities,

they almost complete the full range of ways in which a particular sort of AE can appear in a particular sort of context. (The one remaining case is that of an identifier occurring as a λ-body, which occurs in a later example.)

The right-hand side of an auxiliary definition might itself be qualified by an auxiliary definition, e.g.

| | |
|---|---|
| $u/(u+5)$ | $\{\lambda u.u/(u+5)\}$ |
| **where** $u = a(a+1)$ | $[\{\lambda a.a(a+1)\}[7-3]]$. |
| **where** $a = 7-3$ | |

In particular this might happen with an auxiliary definition of a function, e.g.

| | |
|---|---|
| $f(3)+f(4)$ | $\{\lambda f.f(3)+f(4)\}$ |
| **where** $f(x) = ax(a+x)$ | $[\{\lambda a.\lambda x.ax(a+x)\}$ |
| **where** $a = 7-3$ | $[7-3]]$. |

This last example contains a λ-expression whose body is another λ-expression. Notice that such a λ-expression describes a "function-producing" function and hence can meaningfully give rise to a combination whose operator is a combination, e.g.

$$\{\{\lambda a.\lambda x.ax(a+x)\}[7-3]\}[3].$$

We shall slightly abbreviate such expressions by omitting brackets round an operator that is a combination, i.e.

$$\{\lambda a.\lambda x.ax(a+x)\}[7-3][3].$$

This amounts to an "association to the left" rule. We also abbreviate by omitting brackets round a single identifier,

$$\{\lambda a.\lambda x.ax(a+x)\}[7-3]3.$$

Similarly we may write '$fa+f3+Dfb$' for '$f(a)+f(3)+\{D(f)\}[b]$,' and rely on context to distinguish between '$f$ applied to $a$' and '$f$ times $a$' (as indeed we also do when writing '$f(a+1)$').

Since we shall use multicharacter identifiers (excluding spaces), this abbreviation means that the reader will sometimes be obliged to use his intelligence, together with the context, to decide whether, e.g.

*prefix x nullist*

is to be read as

$$\{prefix[x]\}[nullist]$$

or

$$\{prefi[x][x]\}[nul[list]]$$

or many other conceivable alternatives. Generally we shall use spaces wherever they are helpful without being ungainly in appearance.

We now turn to three forms of expression that play an important role in programming languages, namely lists (in particular argument-lists), conditional expressions and recursive definitions. The next three Sections are devoted to showing how these can be rendered as operator/operand combinations using certain basic functions.

**Lists**

In an earlier Section we gave a structure definition for AEs that made no explicit provision for *lists* of operands. Our illustrations have begged this issue by using dyadic and triadic functions. It will turn out below that discussion of the evaluation of AEs can be simplified if we can avoid classifying operands into "single operands" and "operand-lists," and avoid classifying functions into those that take one argument and those that take several. We now show how this is done.

Lists can be characterized by a structure definition as follows:

A *list* is either *null*
or else has a *head* ($h$)
and a *tail* ($t$) which is a list.

A null-list has length zero. A non-null-list has length one or more; if its items are $a_1, a_2, \ldots, a_k$, $(k \geqslant 1)$, then its head is $a_1$ and its tail is the (null or non-null) list whose $k-1$ items are $a_2, \ldots, a_{k-1}$ and $a_k$. So we let

$$1st = h$$
$$2nd\ L = h(tL)$$
$$3rd\ L = h(t(tL)), \text{ etc.}$$

defining the functions $1st$, $2nd$, etc., in terms of the selectors $h$ and $t$. So the "items" of a list are the things that result from applying $1st$, $2nd$, etc., to it.

On the lines mentioned earlier, the two identifiers *constructnullist* and *constructlist* designate constructors for lists, taking respectively zero and two arguments. So the following equations hold:

$$null(constructnullist(\,)) = \textbf{true}$$
$$null(constructlist(x, L)) = \textbf{false}$$
$$h(constructlist(x, L)) = x$$
$$constructlist(hL, tL) = L$$

and several more.

We shall not distinguish between lists in this sense and the argument lists of dyadic and triadic functions. That is to say, we consider a triadic function to be a function whose arguments are limited to lists of length three. So an operator denoting a triadic function is not necessarily prefixed to an operand-list of three items; e.g. if $L$ is a list of two numbers, the following expression is acceptable:

$$\{\lambda(x, y, z).x+y+z\}[constructlist(3, L)].$$

We use *nullist* to designate a list of length zero, and consider an empty bracket pair as merely an alternative way of writing *nullist*. Also we consider commas as merely an alternative way of writing a particular sort of combination, which we now explain.

Associated with any object $x$ there is a function that transforms any given list into a list of length one more, by adding $x$ at the beginning of it. We denote this function by

$$prefix(x).$$

So if $L$ is a list whose $k$ items are $a_1, a_2, \ldots, a_k$, then

$$prefix(x)L$$

denotes a list whose $k + 1$ items are $x, a_1, \ldots, a_k$. The function *prefix* is function-producing and so gives rise to combinations whose operators are combinations. It can be defined in terms of *constructlist* as follows:

$$prefix(x) = \lambda L.constructlist(x, L).$$

By a natural extension of the notation for function definitions this can also be written

$$prefix(x)(L) = constructlist(x, L).$$

The following examples illustrate the applicative structure we are now imposing on operand-lists of length two or more, and of length zero:

| | |
|---|---|
| $f(a, b, c)$ | $f(prefix\ a(prefix\ b(prefix\ c\ nullist)))$ |
| $a + b$ | $+(prefix\ a(prefix\ b(nullist)))$ |
| $constructnullist()$ | $constructnullist(nullist)$. |

Notice that while it is meaningful to ask whether a function is dyadic (i.e. has arguments restricted to lists of length two), there is no significance to asking whether a function is monadic since any function may be denoted in combination with a single operand rather than a list of operand expressions.

For the rare cases in which we wish to refer to a list with just one item, we use the function defined as follows:

$$unitlist(x) = prefix\ x\ nullist.$$

We shall use the following abbreviation for '*prefix x L*':

$$x : L.$$

So, e.g.

$$x, y, z = x : (y, z) = x : (y : unitlist\ z) = x : (y : (z : ())).$$

We shall treat ':' as more "binding" than ',', e.g.

$$2nd(2nd(L,\ x{:}M,\ N)) = 1st\ M.$$

The last example refers to a list whose items include a list. We admit this possibility and write, e.g.

$$(a, b), (c,(\ ), e), unitlist\ f.$$

In what follows, a list whose items include lists (i.e. a list which has items that are amenable to *null*, *1st*, *2nd*, etc.) will be called a "list-structure."

### Conditional expressions

We now show how AEs provide a match for conditional expressions, e.g. for

$$\textbf{if } a < b \textbf{ then } a^7 \textbf{ else } b^7. \qquad \text{(A)}$$

This expression somewhat resembles

$$i^{\text{th}}(a^7, b^7)$$

where $i$ is a computed index number, used to select an item from a list which is not referred to elsewhere. So, we consider '*if*' to be an identifier designating a function-producing function such that

$$if\ (\textbf{true}) = 1st$$
$$if\ (\textbf{false}) = 2nd.$$

Then (A) is equivalent to the following AE:

$$if\ (a < b)(a^7, b^7). \qquad \text{(A1)}$$

This rendering is not, however, adequate. For it would match

$$\textbf{if } a = 0 \textbf{ then } 1 \textbf{ else } 1/a \qquad \text{(B)}$$

by

$$if\ (a = 0)(1, 1/a). \qquad \text{(B1)}$$

But the value of this expression, i.e. to be more explicit, of

$$if\ (a = 0)(prefix\ 1(prefix(1/a)(\ ))) \qquad \text{(B1')}$$

depends on the value of the sub-expression '$1/a$,' and hence only exists if $1/a$ exists. So (B1) is not an acceptable rendering of (B) if $a$ is zero and division by zero is undefined. More generally, this method of rendering conditional expressions as AEs does not meet our criterion of semantic acceptability unless the domain of every function is artificially extended to contain any argument that might conceivably arise on either "branch" of a conditional expression. We now present another method that avoids any such commitment.

Consider instead the following alternative

$$if\ (a = 0)(\lambda x.1,\ \lambda x.1/a)(3) \qquad \text{(B2)}$$

where '$x$' is an arbitrarily chosen variable and '3' is an arbitrarily chosen operand. Unlike (B1), (B2) has a value even if $a = 0$; for, $\lambda x.1/a$ denotes a function even if $a = 0$ (albeit with null domain—this is in accordance with our view of the "value" of an expression, as introduced informally in a previous Section and formalized in a subsequent one). So (B2) is precisely equivalent to (B) in the sense that either they are equivalent or they are both without value.

The arbitrary '$x$' and '3' in (B2) can be obviated.* For the *bv* of a $\lambda$-expression can be a list of identifiers, and in particular a list whose length is zero. Such a $\lambda$-expression is applicable to an argument list of the same length. This suggests that all conditional expressions can be rendered in a uniform way as follows:

| | |
|---|---|
| $\textbf{if } a < b \textbf{ then } a^7 \textbf{ else } b^7$ | $if\ (a < b)(\lambda(\ ).a^7, \lambda(\ ).b^7)(\ )$ |
| $\textbf{if } a = 0 \textbf{ then } 1 \textbf{ else } 1/a$ | $if\ (a = 0)(\lambda(\ ).1, \lambda(\ ).1/a)(\ ).$ |

### Recursive definitions

The use of self-referential, or "circular," or what have come to be called in the computer world, "recursive" definitions can also be rendered in operator/

* The device given here was suggested by W. H. Burge.

operand terms. By a *circular* definition we mean an implicit definition having the form

$$x = \ldots x \ldots x \ldots x \ldots$$

i.e. a definition of $x$ in which $x$ is referred to one or more times in the definiens. For example, suppose '$M$' designates a list-structure, then

$$(a, M, (b, c))$$

denotes a list-structure whose second item is the list-structure $M$. The equation

$$L = (a, L, (b, c))$$

is satisfied by the "infinite" list-structure containing three items, of which the first is $a$, the third is $(b, c)$ and the second is the infinite list-structure whose first item is $a$, and whose third item is $(b, c)$ and whose second, . . . and so on.

So the above equation may be considered as a circular definition that associates this "infinite" list-structure with the identifier '$L$.'

Again

$$f(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1)$$

i.e.

$$f = \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1)$$

may be considered as a circular definition of the factorial function. (In this brief discussion the important question of whether each circular definition characterizes a *unique* object will be skipped.)

Making use of $\lambda$, any circular definition can be rearranged so that there is just *one* self-referential occurrence, and moreover so that the single occurrence constitutes the operand of the definiens, e.g.

$$L = (a, L, (b, c)) \qquad L = \{\lambda L'.(a, L', (b, c))\}L$$

$$f(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1) \qquad f = \{\lambda f'.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf'(n - 1)\}f.$$

Notice that, had we used '$L$' and '$f$' instead of '$L'$' and '$f'$' they would still have been bound and so would not have constituted self-referential occurrences.

A circular definition of the form

$$x = Fx$$

(such as the last two above) characterizes an object as being invariant when transformed by the function $F$, i.e. as the "fixed-point" of $F$. If we use '$Y$' to designate the function of finding the fixed-point of a given function, such a circular definition can be rearranged so that it is formally no longer circular:

$$x = YF.$$

Thus the above examples become

$$L = (a, L, (b, c)) \qquad L = Y\lambda L.(a, L, (b, c))$$

$$f(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1) \qquad f = Y\lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1).$$

Notice that, according to the above treatment of conditional expressions, the existence of $f(0)$ does not involve the existence of $f(-1)$. Notice also that $Y$ may produce a function, and hence gives rise to combinations whose operators are combinations, e.g.

$$\{Y\lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1)\}6$$

is a meaningful combination. In fact its value is 720.

This device can also be used for a group of "jointly circular" or "simultaneously recursive" definitions, e.g.

$$fx = F[f, g, x] \quad \textbf{and } gx = G[f, g, x] \qquad (f, g) = Y\lambda(f, g).(\lambda x.F[f, g, x], \lambda x.G[f, g, x]).$$

So the fixed-point of a function might be a list of functions. This gives rise to the possibility that a dyadic function might appear with what looks like one, rather than two, arguments, e.g. when the above jointly circular functions appear in an auxiliary definition:

$$f(ga) + g(fb) \textbf{ where } fx = F(f, g, x) \textbf{ and } gx = G(f, g, x) \qquad \{\lambda(f, g).f(ga) + g(fb)\}[Y\lambda(f, g).(\lambda x.F(f, g, x), \lambda x.G(f, g, x))].$$

Notice that the circularity is explicitly indicated in the right-hand version, whereas the left-hand version is only recognizable as circular by virtue of our comments about it or by common sense. In the next Section we shall extend our hitherto informal use of **where** so as to provide a match for any use of $\lambda$.

### The difference between structure and written representation

Our notation for AEs is deliberately loose. There are many ways in which we can write the same AE, differing in layout, use of brackets and use of infixed as opposed to prefixed operators. However, they are all written representations of the *same* AE, in the sense that the information elicited by the questions Q1, Q2 and Q3 above are the same in each case. This is the essential information that characterizes the AE. We call this information the "structure" of the AE. Our laxity with written representations is based on the knowledge that any expression we write could, at the cost of legibility, have been written in standard form, with exclusively prefixed operators and every bracket in place.

One of the syntactic liberties that we shall take is to use **where** instead of $\lambda$. More precisely, we shall use an expression of the form

$$L \textbf{ where } X = M$$

as a "syntactic variant" of

$$\{\lambda X.L\}[M]$$

even in cases that go rather further than the familiar use of **where**, e.g.

$$n^2 + 3n + 2 \textbf{ where } n = n + 1 \qquad \{\lambda n.n^2 + 3n + 2\}[n + 1]$$

$$xy(x + y) \textbf{ where } x = a^2 + a\sqrt{y} \textbf{ where } y = a^2 + b^2 \qquad \{\lambda y.\{\lambda x.xy(x + y)\}[a^2 + a\sqrt{y}]\}[a^2 + b^2].$$

We use indentation to indicate that the **where** qualifies a sub-expression, e.g. in each of the following examples '$y$' occurs both bound and free:

| | |
|---|---|
| $xy(x+y)$<br>**where** $x = a^2 + a\sqrt{y}$<br>**and** $y = a^2 + b^2$ | $\{\lambda(x, y).xy(x+y)\}$<br>$[a^2 + a\sqrt{y}, a^2 + b^2]$ |
| $xy(x+y)$<br>**where** $x = a^2 + a\sqrt{y}$<br>**where** $y = a^2 + b^2$ | $\{\lambda x.xy(x+y)\}$<br>$[\{\lambda y.a^2+a\sqrt{y}\}[a^2+b^2]]$. |

The **where** notation can be extended to allow for circular definitions and jointly circular definitions, thus formalizing a feature of auxiliary definitions that has previously required verbal comment. An occurrence of '$Y$' is indicated by the word **recursive** or, more shortly, **rec**.

| | |
|---|---|
| $f(7-3)$<br>**where rec** $f(n) =$<br>**if** $n = 0$ **then** 1<br>**else** $nf(n-1)$ | $\{\lambda f.f(7-3)\}$<br>$[Y\lambda f.\lambda n.$**if** $n = 0$ **then** 1<br>**else** $nf(n-1)]$. |
| $f(ga) + g(fb)$<br>**where rec** $fx = F(f, g, x)$<br>**and** $gx = G(f, g, x)$ | $\{\lambda(f, g).f(ga) + g(fb)\}$<br>$[Y\lambda(f, g).(\lambda x.F(f, g, x),$<br>$\lambda x.G(f, g, x))]$. |

It will be observed that our discussion of applicative structure has doubled back on itself. We started by remarking the possibility of analyzing certain more or less familiar notations in terms of functional application and functional abstraction. We are now remarking the possibility of looking upon these notations as "really" AEs, written with syntactic variations that make them more palatable. Clearly, once a semantically acceptable correspondence between AEs and some other notation has been established, it can be looked at in either way.

The above explanation of **where** and AEs leaves some details unsettled, but should be enough to make the use of **where** in what follows (*a*) comprehensible and (*b*) plausibly a mere "syntactic sugaring" of AEs. Further discussion of **where**, or of other sorts of syntactic sugar, is outside the scope of this paper.

Another example of alternative notations concerns conditional expressions. Interchangeably with the **if** . . . **then** . . . notation we use the $\rightarrow$ notation as illustrated by the following two examples:

| | |
|---|---|
| $p \rightarrow a$<br>**else** $\rightarrow b$ | **if** $p$ **then** $a$<br>**else** $b$. |
| $p \rightarrow a$<br>$q \rightarrow b$<br>**else** $\rightarrow c$ | $p \rightarrow a$<br>**else** $\rightarrow (q \rightarrow b$<br>**else** $\rightarrow c)$. |

Any particular set of rules about representing AEs by written text (the correspondence with **where** is one such set of rules) has two aspects:

(*a*) a rule for deriving the structure of an AE, given a text that represents it,
(*b*) a rule for deriving a text that represents an AE, given its structure.

The formalization of these rules, and in particular their formalization as AEs, is another topic that is outside the scope of this paper.

### The power of applicative expressions

We have described how certain expressions can be considered as being constructed from "known" identifiers, or "constants," by means of functional application and functional abstraction. We might look at the situation another way round and consider how many expressions can be constructed, starting with a given selection of constants, using these same means of construction. More precisely, we might compare working within such constraints to working within some other set of constraints, e.g. some algebraic programming language or machine code, or system of formal logic. It transpires that the seven objects, *null*, *h*, *t*, *nullist*, *prefix*, *if* and *Y* provide a basis for describing a wide range of things.

Roughly speaking, when taken together with functional application and functional abstraction, they perform the "house-keeping" or "red-tape" roles that are performed by sequencing, indices and copying in a conventional programming language, and by narrative in informal mathematics. For example:

(1) With a few basic numbers and numerical functions they are sufficient to describe the numbers and functions of recursive number theory. So they are in some sense "as powerful as" other current symbolisms of mathematical logic and computer programming. The question whether this sense has much practical significance is one that will not be discussed here.
(2) With a few basic symbols, and functions associated with classes of symbol-strings, they are sufficient to describe syntax (of, say, ALGOL 60, or for AEs themselves), from the point of view both of synthesizing and of analyzing.
(3) With a few basic classes, and functions associated with classes of composite information structures, they are sufficient to formalize "structure definitions," as introduced above (for example the structure definition of AEs themselves).
(4) With a few structure definitions they are sufficient to characterize formally the "value" of an AE, and to describe a mechanical process for "producing" it. This is the use to which AEs will be put in the rest of this paper.

A discussion of the relative convenience of various notations in the fields mentioned here is outside the scope of this paper.

## Evaluation

### The value of an applicative expression

Every AE in the above examples, including every sub-expression of every AE, has a "value," which is either a number, or a function, or a list of numbers, or a list

of functions, etc. More precisely, an AE $X$ has a value (or rather *might* have a value) relative to some background information that provides a value for each identifier that is free in $X$. This background information will be called the *environment* relative to which evaluation is conducted. It will be considered as a function that associates with each of certain identifiers, either a number, or a list, or a function, etc. Each identifier to which an environment $E$ gives a value is called a *constant* of $E$, and each object "named," or "designated," by a constant of $E$ (possibly by several) is called a *primitive* of $E$. So $E$ is a function whose domain and range comprise respectively its constants and its primitives.

If we let

$$val(E)(X)$$

denote the value of $X$ relative to $E$ (or *in* $E$ for short), the function that *val* designates can be specified by means of three rules, R1, R2 and R3. These correspond to the three questions, Q1, Q2 and Q3 that were introduced earlier to elucidate the structure of AEs.

R1. If $X$ is an identifier, $valEX$ is $EX$;
(R2. appears below);
R3. If $X$ is a combination, $valEX$ can be found by first subjecting both its operator and operand to $valE$, and then applying the result of the former to the result of the latter.

The rules R1 and R3 are enough to specify $valEX$ provided that $X$ contains no λ-expressions. For example, consider an environment in which the identifier $k$ is associated with the number 7 and the identifier $p$ with the truthvalue **false**, and other identifiers have their expected meanings. Then R1 and R3 suffice to fix the value of, say,

$$if((2^{19} < 3^{12}) \vee p)(sin, cos)(\pi/k).$$

This example illustrates the need for evaluating the operator of a combination as well as its operand.

R2. If $X$ is a λ-expression, $valEX$ is a function. Like any function it can be specified by specifying what result it produces for an arbitrary argument, and we now do this as follows: $valEX$ is that function whose result for any given argument can be found by evaluating $bodyX$ in a new environment derived from $E$ in a way we shall presently describe. For example, suppose $E$ is the environment postulated above, and $X$ is the λ-expression '$\lambda r.k^2 + r^2$.' Then its value in $E$ is that function whose result for any given argument, say 13, can be found by evaluating '$k^2 + r^2$' in a new environment $E'$, derived from $E$. To be precise, $E'$ agrees with $E$ except that it gives the value 13 to the identifier $r$.

More generally, this derived environment consists of $E$, modified by pairing the identifier(s) in $bvX$ with corresponding components of the given argument $x$ (and using the new value for preference if any variable in $bvX$ coincides with a constant of $E$). We denote this derived environment by

$$derive(assoc(bvX, x))E.$$

We shall describe below a mechanical process for obtaining the value, if it exists, of any given AE relative to any given environment. This process can be implemented with pencil and paper, or (as we shall briefly sketch) with a digital computer. The rules R1 and R3 provide a criterion for deciding whether or not the outcome of this process is in fact the value as we understand it.

The three rules can be formalized as a definition of *val*, thus:

**recursive** $valEX = identifierX \rightarrow EX$
$\lambda expX \rightarrow f$
**where** $fx = val(derive(assoc(bvX,x))E)$
$(bodyX)$
**else** $\rightarrow \{valE(ratorX)\}[valE(randX)]$.

For example, suppose *thrice* is the function-producing function defined by

$$thrice(f)(x) = f(f(f(x))).$$

Then it follows from the above definition of *val* that the values of the following five AEs,

*square* 5
*thrice square* 5
*thrice square* (*thrice square* 5)
*thrice* (*thrice square*) 5
*thrice thrice square* 5

are respectively $5^2$, $5^{2^3}$, $5^{2^6}$, $5^{2^9}$ and $5^{2^{27}}$. The reader may be better equipped to check this assertion when he has read the next Section, which describes an orderly way of evaluating AEs.

The set of objects that can be denoted by an AE relative to an environment $E$, is the range of the function $valE$. It contains all the primitives of $E$, and everything produced by such an object, and every function that can be denoted by a λ-expression.

## Mechanical evaluation

In order to mechanize the above rule, we represent an environment by a list-structure made up of name-value pairs. There is a function designated by *location* such that if $E^*$ is this structure and $X$ is an identifier then

$$locationE^*X$$

denotes the selector that selects the value of $X$ from $E^*$. So if $E^*$ represents the environment $E$ then the following equation holds:

$$valEX = locationE^*XE^*.$$

We shall not bother below to distinguish between $E$ and $E^*$.

Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

The value relative to $E$ of a λ-expression $X$ is represented by the closure denoted by

$$constructclosure((E, bvX), unitlist(bodyX)).$$

This particular arrangement of the information in a closure has been chosen for our later convenience.

We now describe a "mechanization" of evaluation in the following sense. We define a class of COs, called "states," constructed out of AEs and their values; and we define a "transition" rule whose successive application starting at a "state" that contains an environment $E$ and an AE $X$ (in a certain arrangement), leads eventually to a "state" that contains (in a certain position) either *valEX* or a closure representing *valEX*. (We use the phrase "result of evaluation" to cover both objects and closures. We suppose that the identifier *closure* designates a predicate that detects whether or not a given result of evaluation is a closure.)

A *state* consists of a *stack*, which is a list, each of whose items is an intermediate result of evaluation, awaiting subsequent use;

and an *environment*, which is a list-structure made up of name/value pairs;

and a *control*, which is a list, each of whose items is either an AE awaiting evaluation, or a special object designated by '*ap*,' distinct from all AEs;

and a *dump*, which is a complete state, i.e. comprising four components as listed here.

We denote a state thus:

$$(S, E, C, D).$$

The environment-part (both of states and of closures) would be unnecessary if λ-expressions containing free variables were prohibited. Also the dump would be unnecessary if all λ-expressions were prohibited.

Each step of evaluation is completely determined by the current state $(S, E, C, D)$ in the following way:

1. If $C$ is null, suppose the current dump $D$ is

$$(S', E', C', D').$$

Then the current state is replaced by the state denoted by

$$(hS : S', E', C', D').$$

2. If $C$ is not null, then $hC$ is inspected, and:

(2*a*) If $hC$ is an identifier $X$ (whose value relative to $E$ occupies the position *locationEX* in $E$), then $S$ is replaced by

$$locationEXE : S$$

and $C$ is replaced by $tC$. We describe this step as follows: "Scanning $X$ causes *locationEXE* to be loaded."

(2*b*) If $hC$ is a λ-expression $X$, scanning it causes the closure derived from $E$ and $X$ (as indicated above) to be loaded on to the stack.

(2*c*) If $hC$ is *ap*, scanning it changes $S$ as follows: $hS$ is inspected and:

(2*c*1) If $hS$ is a closure, derived from $E'$ and $X'$, then: $S$ is replaced by the nullist,
$E$ is replaced by
$derive(assoc(bvX', 2nd\,S))E'$,
$C$ is replaced by $unitlist(bodyX')$,
$D$ is replaced by $(t(tS), E, tC, D)$.

(2*c*2) If $hS$ is not a closure, then scanning *ap* causes $S$ to be replaced by
$((1st\,S)(2nd\,S) : t(tS))$.

(2*d*) If $hC$ is a combination $X$, $C$ is replaced by $randX : (ratorX : (ap : tC))$.

Formally this transformation of one state into another is

```
Transform(S,E,C,D) =
  nullC → [hS:S',E',C',D']
          where S',E',C',D' = D
  else →
    identifierX → [locationEXE:S, E, tC, D]
    λexpX →
      [constructclosure((E, bvX),unitlist(bodyX)):S,
      E, tC, D]
    X = ap → closure(hS) →
                  [( ), derive(assoc(J,2ndS)E'),
                       C,
                       (t(tS), E, tC, D)]
                  where E',J = environmentpart(hS)
                  and C' = controlpart(hS)
                  else → [(1stS)(2nd):t(tS), E, tC, D]
    else → [S, E, randX:(ratorX:(ap:tC)), D]
    where X = hC
```

We assume here that an AE composed of a single identifier is the same object as the identifier itself. This suggests a more general assumption that whenever one of the alternative formats of a structure definition has just one component, the corresponding selector and constructor are both merely the identity function. We also assume that a state is identical to a list of its four components. This suggests a more general assumption that whenever a structure definition allows just one format, the constructor is the identity function. Without these assumptions the above definition would be a bit more

elaborate. A formal account of structure definitions would lead to a more careful discussion of these points.

Notice that, whereas a previous formula described a rule for deriving from an AE its value, this new formula describes a rule for advancing a certain information-structure through one step. If $X$ is an AE, and $E$ is an environment such that *valEX* is defined, then starting at any state of the form

$$S, E, X : C, D$$

and repeatedly applying this transformation, we shall eventually reach the state denoted by

$$valEX : S, E, C, D.$$

That is to say, at some later time $X$ will have been scanned and its value relative to the current environment will have been loaded (on to the stack). In particular, if $S$ and $C$ are both null, i.e. if the initial state is

$$(), E, unitlistX, (S', E', C', D')$$

there will be a subsequent state

$$unitlist(valEX), E, (), (S', E', C', D')$$

which will be immediately succeeded by the state denoted by

$$valEX : S', E', C', D'.$$

These assertions can be verified by performing the appropriate substitutions in the definition of *Transform*.

**Basic functions**

By a "basic" function of $E$ we mean a function other than a closure, that can arise as a result of evaluation. At the most the basic functions comprise

(1) primitive functions;
(2) any functions produced by basic functions.

For, any result of a closure must also be a result of a primitive (or be a result of a result of a primitive, or, etc.).

However, this may be an over-estimate of the number of basic functions, for it is clearly possible that a primitive might be a closure. For instance the evaluation of

$$\{\lambda f.f3 + f4\}[\lambda x.x^2 + 1]$$

relative to $E$ involves evaluating

$$f3 + f4$$

relative to an environment in which '$f$' names the closure that we may roughly denote by

$$constructclosure((E, \text{'}x\text{'}), unitlist(\text{'}x^2 + 1\text{'})).$$

Of the six sorts of step described above, namely (1), (2*a*), (2*b*), (2*c*1), (2*c*2) and (2*d*), all except (2*c*2) are mere rearrangements. (2*c*2) arises whenever *ap* finds that the head of the stack is a basic function.

**Other ways of mechanizing evaluation**

It should be observed that this is only one of many ways of mechanizing the evaluation of AEs, all producing the value, as specified above. For instance, it is not essential that the operand of a combination be evaluated before its operator. The operand might be evaluated after the operator; it might even be evaluated piecemeal when and if it is required during the application of the value of the operator. Again, the evaluation of a $\lambda$-expression might be accompanied by partial evaluation of its body. The AE might be subjected to pre-processing of various kinds, e.g. to disentangle combinations once for all or to remove its dependence on an arbitrary choice of identifiers occurring bound in it. The pre-processing might be more elaborate and perform symbolic reduction.

The particular evaluation process described above will be called *normal* evaluation, and its significance partly lies in that many other evaluation processes can be described in terms of it; i.e. they can be specified as a transformation of the AE into some other AE, followed by normal evaluation of the derived AE. Further discussion of evaluation processes and of their mutual relationships is outside the scope of the present paper.

**Evaluating with a digital computer**

This Section describes how a "state," in the above sense, can be represented in the instantaneous state of a digital computer, and how the transformation formalized above can be represented by a stored program. The method chosen here is one of many and is distinguished by its simplicity in description, rather than by its cheapness. It will hold no surprises for anyone familiar with the "chaining" techniques of storage and location pioneered in list-processing systems. It is given here as a demonstration of possibility, not of practicability.

*Representing each composite object by an address*

Each component of a state, from the entire state downwards, and including such COs as are definable objects, can be represented in a computer by an address. The way of doing this is closely related to the structure definitions used to introduce the various COs concerned. For, given that the components can be represented by addresses, the complete CO can be represented by a short segment of store, large enough to contain these addresses (and, if the CO is one admitting alternative formats, a distinguishing tag). So the complete CO can also be represented by the *address* of this short segment. There is need for one fixed area in the store, large enough to hold an address and representing *the* current state. The merit of this method is that the predicates, selectors, and constructors can be represented by stored programs whose size and speed is independent of the size of the COs operated on. Hence this is also true of the information-rearranging steps that occur during evaluation, namely (1), (2*a*), (2*b*), (2*c*1) and (2*d*); for each of these is a composition of predicates, selectors and constructors.

Each of these steps can be represented by a stored program of ten or twenty orders in most current machine-codes. Obviously, the possibility arises of designing a machine-code that favours these steps.

However, the implementation sketched here has less claim to such embodiment than some others whose properties are briefly referred to below.

*Shared components*

One consequence of this method is the presence of "shared" components. For instance, suppose the environment denoted by

$$derive(assoc(bvX, x))E$$

is being formed in step (2*c*1). It is possible that a copy of the address representing E is "incorporated" into the new environment. As long as environment components are not updated, the extent of sharing is immaterial. However, there are two possible developments in which it would become important to consider precisely what components are shared.

(*a*) We might vary the evaluation process by introducing a preparatory investigation of each AE, to determine whether any of the transformations of COs that occur during its evaluation can be performed by overwriting rather than by reconstructing.
(*b*) We might generalize AEs by introducing a fourth format playing the role of an assignment.

*Representing each non-composite object by an address*

The possibility of using the above storage technique depends on

(1) being able to represent each *non-composite* definable object by an address: namely, identifiers, primitives and all results of evaluation other than closures and composite definable objects;
(2) being able to represent each basic function $f$ by a stored program such that, if the head of the stack represents $x$, the program replaces it by an address representing $fx$.

*Representing Y*

If we consider a specific (powerful) set of primitives, comprising some basic numbers, some numerical functions, the basic list-processing primitives, and $Y$, only the latter involves any unfamiliar technique. $Y$ can be represented by a stored program that, given an argument $F$ at the head of the stack, performs the following steps:

1. Take a fresh cell $z$, whose address is $Z$.
2. Use $Z$ as a spurious argument for $F$, producing a result-address $Z'$.
3. Copy the word addressed by $Z'$ to the cell $z$. Then $Z$ is the required result of $Y$.

This representation of $Y$ is adequate for the uses of it mentioned in the Section on "Circular definitions."

*Source of storage*

The stored programs for constructing COs must have access to a source of fresh storage cells, which (unless the machine is to be congested rapidly) must in turn be able to retrieve for re-use used cells that have become irrelevant.

*Other ways of representing our mechanization with a digital computer*

It was earlier observed that the mechanization in terms of SECD-states is only one of many ways of mechanizing evaluation. Likewise, given a particular mechanization, there may be many ways of representing it with a digital computer. In particular, the method just sketched is not the only way of mechanizing SECD-states.

For example, of all the occasions on which a fresh cell is required, there are certain sub-sets that can reasonably be acquired and disposed of in a "last in/first out" (LIFO) pattern. Hence by distributing these requirements among more than one source of fresh cells it is possible to exploit consecutive addressing. In particular by restricting the structure of AEs it is possible to rely exclusively on the LIFO pattern. Such restrictions suggest a pre-evaluational transformation for eliminating expensive structures in favour of equivalent cheaper ones. Such variations are outside the scope of this paper.

## Conclusion

Several lines of development outside the scope of this paper have been indicated above. Some of these consist in part of a "sideways advance," a rephrasing of previous work in a new jargon. However, a new jargon might have features that justify this procedure. The best claim in the present case seems to be based on the extent to which it isolates several aspects of the use of computers that frequently appear inextricably interwoven.

One such feature is the distinction between structure and representation effected by "structure definitions." For instance, the structure of expressions was distinguished from their written representation. Again, the structure of the information that is recorded during evaluation was distinguished from its representation in a computer.

Another separation achieved above is that between considerations special to a particular subject-matter, and considerations relevant to every subject-matter (or "universe of discourse," or "field of application," or "problem orientation"). The subject-matter is determined precisely by the choice of primitives and is not affected by the choice of names for them, or of rules for writing expressions (except that these rules might narrow the subject-matter by making some AEs unwritable).

The relationship between expressions and their written representation encompasses all that is customarily called the "syntax" of a language and part of what is customarily called its "semantics." The chosen name/value relation, together with the primitives themselves (that is to say, the applicative relationships between them) constitute the rest of what is customarily called the

"semantics" of a language insofar as it is distinct from the semantics of other languages.

These remarks about "languages" are subject to an important qualification. They apply only to languages that can be considered as AEs plus syntactic sugar. While most languages in current use can partly be accounted for in these terms, entirely "applicative" languages have yet to be proved adequate for practical purposes. Whether or not they will be, and whether their interesting properties can be extended to hold for currently useful languages are questions outside the scope of this paper.

## Relation to other work

Most of the above ideas are to be found in the literature. In particular Church and Curry, and McCarthy and the ALGOL 60 authors, are so large a part of the history of their respective disciplines as to make detailed attributions inevitably incomplete and probably impertinent.

The criterion of "semantic acceptability," whereby a proposed rendering in terms of AEs can be judged correct or incorrect, is closely related to what some logicians (e.g. Quine [8]) call "referential transparency," and to what Curry [2] calls the "monotony" of equivalence.

Structure definitions are in some sense merely a convenient way of avoiding the uninformative strings of a's and d's that occur in LISP's 'cadar,' 'cadaddr,' etc. [6]. However, they have another merit, that of being less associated with a particular internal representation, and, in particular, with a particular ordering of the components. (Gilmore [5] effectively uses "selectors" to avoid entanglement with a specific written representation of expressions.)

Church [1], Curry [2] and Rosenbloom [9] all include discussions of how to eliminate various uses of bound variables in terms of just one use, namely functional abstraction; also of how to eliminate lists, and functions that select items from lists, in terms of functional application. The function $Y$ is called $\Theta$ in Rosenbloom [9], and **Y** in Curry [2]; roughly speaking $Y\lambda$ is McCarthy's *label* [6].

Formalizing a system in its own terms is now a familiar occupation. The relative simplicity of the function *val*, compared, say, with LISP's *eval*, *apply*, etc. [6, 7], is due partly to the fact that it treats the operator and operand of a combination symmetrically.

The formalization of a machine for evaluating expressions seems to have no precedent. Gilmore's machine [5] is specified by a flow diagram. The relative simplicity of the function *Transform*, compared with his specification, is also due in part to the above mentioned symmetry. Closures are roughly the same as McCarthy's "FUNARG" lists [7] and Dijkstra's PARD's [3]. (This method of "evaluating" a $\lambda$-expression is to be contrasted with "literal substitution" such as is used in Church's normalization process, in Gilmore's machine [5], and in Dijkstra's mechanism [4]).

## References


1. Church, A. (1941). *The Calculi of Lambda-Conversion*, Princeton, Princeton University Press.
2. Curry, H. B., and Feys, R. (1958). *Combinatory Logic*, Vol. 1, Amsterdam, North Holland Publishing Co.
3. Dijkstra, E. W. (1962). "An ALGOL60 Translator for the X1," *Automatic Programming Bulletin*, No. 13.
4. Dijkstra, E. W. (1962). "Substitution Processes," Preliminary Publication, Amsterdam, Mathematisch Centrum.
5. Gilmore, P. C. (1963). "An Abstract Computer with a LISP-like Machine Language without a Label Operator," in *Computer Programming and Formal Systems*, ed. Braffort, P., and Hirschberg, D., Amsterdam, North Holland Publishing Co.
6. McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1," *Comm. A.C.M.*, Vol. 3, No. 4, pp. 184–195.
7. McCarthy, J. *et al.* (1962). *LISP* 1.5, *Programmer's Manual*, Cambridge, M.I.T.
8. Quine, W. V. (1960). *Word and Object*, New York, Technology Press and Wiley.
9. Rosenbloom, P. (1950). *The Elements of Mathematical Logic*, New York, Dover.