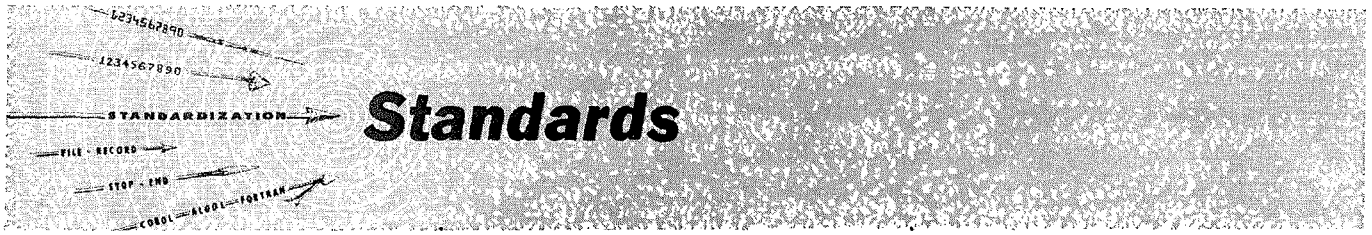

A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I*

By P. J. Landin†

This paper describes how some of the semantics of ALGOL 60 can be formalized by establishing a correspondence between expressions of ALGOL 60 and expressions in a modified form of Church's λ-notation. First a model for computer languages and computer behavior is described, based on the notions of functional application and functional abstraction, but also having analogues for imperative language features. Then this model is used as an "abstract object language" into which ALGOL 60 is mapped. Many of ALGOL 60's features emerge as particular arrangements of a small number of structural rules, suggesting new classifications and generalizations.

The correspondence is first described informally, mainly by illustrations. The second part of the paper gives a formal description, i.e. an "abstract compiler" into the "abstract object language." This is itself presented in a "purely functional" notation, that is one using only application and abstraction.

Contents

Introduction

Anyone familiar with both Church's λ-calculi (see e.g. [7]) and ALGOL 60 [6] will have noticed a superficial resemblance between the way variables tie up with the λ's in a nest of λ-expressions, and the way identifiers tie up with the headings in a nest of procedures and blocks. Some may also have observed that in, say

$$\{\lambda f.f(a) + f(b)\}[\lambda x.x^2 + px + q]$$

the two λ-expressions, i.e. the *operator* and the *operand*, play roughly the roles of block-body and procedure-declaration, respectively. The present paper explores this resemblance in some detail.

The presentation falls into four sections. The first section, following this introduction, gives some motivation for examining the correspondence. The second section describes an abstract language based on Church's λ-calculi. This abstract language is a development of the AE/SECD system presented in [3] and some acquaintance with that paper (hereinafter referred to as [MEE]) is assumed here. The third section describes informally, mainly by illustrations, a correspondence between expressions of ALGOL 60 and expressions of the abstract language. The last section formalizes this correspondence; it first describes a sort of "abstract ALGOL 60" and then presents two functions that map expressions of abstract ALGOL 60 into, on the one hand, ALGOL 60 texts, and on the other hand expressions of the abstract language.

Motivation

It seems possible that the correspondence might form the basis of a formal description of the semantics of ALGOL 60.[1] As presented here it reduces the problem of specifying ALGOL 60 semantics to that of specifying the semantics of a structurally simpler language. The formal treatment of the latter problem is beyond the scope of this paper, and hence likewise a formal proof that the correspondence described here is correct. It is hoped that the informal account of the semantics of the abstract "object language"

* Part II of this paper, which gives the Formal Presentation of the Correspondence, will appear in the March, 1965 issue of the *Communications of the ACM*.

† Present address: Univac Division of Sperry Rand Corporation, Systems Programming Research, New York, New York.

[1] This view is expanded in [10].

will be enough to justify the choice of correspondence in all but a few details.

Long-Term Prospects

There are two ways in which this work might be relevant to the job of providing software.

(1) Formal syntax has been used to practical advantage by language designers and implementers. There might be analogous advantages in formal semantics.

(2) If several languages are being used concurrently by the users of a computer, a common formal basis for them might help in intermixing them (e.g. referring in one language to program that originated in another).

Clearly, significance in these fields depends on the possibility of applying to other languages the technique illustrated here in respect to Algol 60. So far only sketchy trials have been made, since even for Algol 60 the present state of the investigation is not satisfactory. Discussion of other languages is largely beyond the scope of this paper, but the sketchy trials confirm what will be obvious to the present reader—that the method is at its best with languages that rely mainly on elaborate "descriptive" forms, and at its worst with those that rely mainly on elaborate "imperative" forms. Thus it favours what currently tend to be called "advanced" languages—those with good definition facilities, localized naming, and recursive structure of right-hand sides and programs. It has little value with a fixed-format, absolute-address language that makes extensive use of sequencing and of state-indicators that modify the effect of subsequently executed instructions.

Short-Term Aims

In [MEE] it was shown how certain features of programming languages can be modeled in a modified form of Church's λ-notation. More precisely, a model language, based on λ-calculi, and called *applicative expressions* (AEs) was described in abstract terms, that is to say, independently of any particular written representation. It was shown how AEs provide facilities essentially equivalent to auxiliary definitions, conditional expressions and recursion, and that with a suitable choice of written representation these facilities take on a familiar appearance. Furthermore, an abstract machine, called the SECD-machine, was described capable of "interpreting" AEs. Still more precisely, there is a *family* of languages and a matching family of machines, each characterized by an "environment," i.e. a set of named objects that are related to one another by rules concerning the results of applying them one to another. The present paper is based on a development of this scheme, intended to incorporate Algol 60.

The attempt to fit Algol 60 into the AE/SECD framework can be considered from two sides. On the one hand, for someone familiar with Algol 60 it may clarify some of the features of AEs. Firstly, the analysis of Algol 60 in terms of AEs illustrates the distinction made in [MEE], between "applicative structure" and "syntactic sugar." Secondly, we shall give many examples of the use of AEs as a descriptive tool, and in particular some that exhibit in AEs features that correspond to the sequencing control of conventional programming languages. It emerges that the choice of an AE from a certain class of similar AEs is rather like the choice of an order for a set of assignments, and that the issue of whether or not this choice affects the outcome has an analogy in terms of AEs.

For example, it follows from the definition of the "value" of an AE that

$$if(a \leqq 0)(a^2, 1/a)$$

and

$$if(a \leqq 0)(\lambda(\ ).a^2, \lambda(\ ).1/a)(\ )$$

are not necessarily equivalent; if $a = 0$ the first is undefined. This difference is reflected in the behavior of the SECD-machine. In evaluating the first it attempts to evaluate both a^2 and $1/a$, whereas in evaluating the second, both $\lambda(\ )$-expressions are evaluated but only one of their bodies. The use of λ, and in particular (to avoid an irrelevant bound variable) of $\lambda(\ )$, to delay and possibly avoid evaluation is exploited repeatedly in our model of Algol 60. A function that requires an argument-list of length zero is called a *none-adic* function.

On the other hand, AEs illuminate the structure of Algol 60.

Firstly, the definitive description of Algol 60 is split into sections headed "syntax" and "semantics." The syntax is described formally, the semantics mainly by narrative. This paper provides a way of formalizing some of the semantics by associating semantic features of Algol 60 with syntactic features of AEs. (More precisely, we should say *structural* features of AEs, since AEs have an existence independent of any specific written representation.)

Secondly, the analysis leads to a classification of the grammatical units of Algol 60 that provides a unified framework for such features as (i) the fact that array bounds are necessarily specified in terms of nonlocals (and hence, for instance, cannot use a procedure declared in parallel); and (ii) the commonly observed similarity between procedures and actual parameters called by name. More generally it shows that many features of Algol 60, including call by name, declarations, **own, for**-lists, and labels, have parallels in a language that lacks assignment, and are hence not essentially related to "imperative" languages. Also, mainly by default, it suggests a new view of **own.**[2]

Thirdly, the analysis brings out certain respects in which Algol 60 is "incomplete," in that it provides differing facilities in situations that are closely analogous. It dictates what extensions (and small changes) are required to remove this incompleteness. Whether or not "completeness," in the technical sense that emerges below, is a desirable at-

[2] Developed in [10].

tribute of programming language will not be discussed directly in this paper beyond the following remarks.

In its favor it can be said that, given completeness, many of the proposals for extensions that are put forward with reference to current programming languages become mere *syntactic* proposals, i.e. proposals for writing in an allegedly more convenient way some particular kind of expression that is already in the language. There is an important qualification to this: such a proposal might have some economic overtones; i.e. it might be a syntactic proposal that helps the machine to recognize when special "cheap" techniques are appropriate, or it might deter the user from writing forms of expression that are expensive to implement.

The above claim for complete languages sounds like one often made to rebut a proposed language extension—"You can do that already merely by writing ... ." The question therefore arises whether a complete language forestalls its extensions in any more significant sense than do Turing machines, Markov algorithms, built-in instruction codes of general-purpose computers, or programming languages with ALGOL-like procedure facilities. We shall not pursue this question here.

Fourthly, the analysis of ALGOL 60 in terms of AEs suggests a new way of comparing various run-time setups for ALGOL 60, and displays the interactions between run-time setup and the extensions mentioned above.

Imperative Applicative Expressions

We now describe the abstract language used later to model ALGOL 60. We give a formal account of its structure (i.e. its "abstract syntax," to use McCarthy's phrase [5]) and an informal account of its semantics—very much what the official description does for ALGOL 60. The semantics of this abstract language are described in terms of an abstract machine for interpreting it. This language consists of expressions we call *imperative* AEs (IAEs), and the machine is called the *sharing machine*. The IAE/sharing machine system is a development of the AE/SECD system described in [MEE]. However there is an important way in which the relationship of IAEs to the sharing machine differs from that of AEs to the SECD-machine. The semantics of AEs can be specified formally without recourse to a machine. In fact this specification provides a criterion for judging the "correctness" of a machine that purports to evaluate AEs. The SECD-machine satisfies this criterion but it is not the only (abstract) machine to do so. With IAEs on the other hand it appears impossible to avoid specifying semantics in terms of a machine.

A relationship was established in [MEE] between AEs and certain informally introduced pieces of notation; for example, we consider

$$X \quad \textbf{where} \quad x = Z$$

as a way of writing the operator/operand combination whose operand is Z and whose operator is a λ-expression whose bound variable is x and whose λ-body is X; this is the AE that in a more rigid notation is written

$$\{\lambda x.X\}[Z]$$

In this paper we use another piece of "syntactic sugar" for the same AE structure, namely,

$$\textbf{let} \quad x = Z; \quad X$$

or, letting layout obviate punctuation,

$$\begin{array}{l}\textbf{let} \quad x = Z \\ X\end{array}$$

Here is a typical form of AE, presented both informally and formally.

let $a = A$	$\{\lambda(a, b, f).$
and $b = B$	$\quad\{\lambda(g, h).$
and $f(x, y) = F$	$\quad\quad\{\lambda k.X\}[\lambda(u, v).\lambda x.K]\}$
let rec $g(x) = G$	$\quad[Y\lambda(g, h).(\lambda x.G, \lambda(y, z).H]\}$
$\quad$ **and** $h(y, z) = H$	$[A, B, \lambda(x, y).F]$
let $k(u, v)(x) = K$	
X	

The only consideration in choosing between **let** and **where** will be the relative convenience of writing an auxiliary definition before or after the expression it qualifies.

The use of commas for denoting lists, → for conditional expressions, and **let** and **where** for auxiliary definitions, provides a way of representing AEs on paper, i.e. one particular "syntax" for AEs. Though this syntax has not been rigorously defined, enough has been said to ensure that each formula could be rewritten in a rigid, prefixed operator, fully bracketed notation (rewritten, that is to say, in at least one way and possibly several equivalent ways). Another way of saying this is that if we know how to translate ALGOL 60 into a language based on just operator/operand combination, conditional expressions and auxiliary definitions (including function definitions and recursive definitions) then we effectively know how to translate ALGOL 60 into AEs. We shall use this fact to make the informal discussion here less technical than it otherwise would be.

In the formulas of [MEE], AEs play two independent roles. First they are part of the subject matter in the sense that certain subexpressions denote AEs. Second they are the language in use, since all the expressions are to be considered as written presentations of AEs. This dual role is maintained in the present paper. It is fortuitous in the sense that we might have used some other means of expressing what we have to say about AEs, or we might have chosen some other topic and nevertheless have used AEs. However, the coincidence is designed in the sense that our interest in AEs springs partly from their success in describing their own features.

A Generalization of Jumps

We introduce into the SECD-machine an operation denoted by 'J', that is applicable to functions, or more pre-

cisely to *closures*, and modifies their subsequent exit behavior.

For example, consider a definition of f as follows:

$$f(x) = \cdots g(\cdots, \cdots)\cdots$$
$$\textbf{where } g = \mathrm{J}\lambda(u, v).$$

where the subexpression $g(\cdots, \cdots)$ may occur at any λ-depth within the right-hand side, and in any context. If during an application of f this subexpression is evaluated, its value will immediately become the result produced by f. This is reminiscent of an alarm exit from a subroutine. In fact the generalization of labels to permit arguments may be a useful way of providing for alarm exits in ALGOL-like languages.

Further, to define precisely the meaning of 'J', we extend the class of possible results of evaluation by introducing a new kind of object, a bundle of information called a "program-closure." When J is applied to a closure it transforms into a program-closure.

The resulting program-closure, like a closure, includes the current environment (for subsequent installation) and an expression (for subsequent evaluation). However it also includes the current dump, and when the bundle eventually comes to be activated, this dump is also installed. This process is to be contrasted with the activation of a closure, in which the dump is used to record the current state in order that it may be resumed later.

We call an expression of the form

$$\mathrm{J}(\lambda L \cdot S)$$

a "program-point."[3] Roughly speaking, ALGOL 60's labels are a special case of program-points. They are parameterless, and the λ-body is typically a functional product whose terms correspond to the statements following the label. Moreover, references to both closures and program-closures are restricted in ALGOL 60 in a way that prevents them from being carried outside the scope in which they are produced. (In LISP [4] references to closures are free of this restriction, hence the need for a chained stack. However program-closures are more severely constrained in that they cannot be carried into an inner scope.)

INTRODUCING COMMANDS INTO A FUNCTIONAL SCHEME

Any attempt to fit most current programming languages into the AE/SECD scheme involves the questions: Are commands to be construed as subexpressions and, if so, what do they denote? In particular, what does an assignment denote and what does a jump denote? We postpone these questions here (although claiming to leave them rather more clearly formed than we found them) by using another family of languages similar to the one associated with the AE/SECD system but differing as follows.

The notion of AE is extended to comprise one new format that models assignment. We characterize the enlarged set as follows.

An imperative applicative expression (IAE) is
  either an *identifier*,
  or a λ-*expression* (λexp) and consists of
    its *boundvariable* part (*bv*), which is a list-structure of identifiers, and
    its λ-*body* (*body*), which is an IAE,
  or an *assigner*, which consists of
    its *lefthandside* (*lhs*), which is an IAE, and
    its *righthandside* (*rhs*), which is an IAE,
  or a *combination*, which consists of
    its *operator* (*rator*), which is an IAE, and
    its *operand* (*rand*), which is an IAE.

We adopt, informally, the following notation for assigners,

$$\text{lhs} \Leftarrow \text{rhs}$$

This extension of the notion of an expression brings in its train the problem of extending the notion of *the meaning of* an expression. The next subsection is an informal approach to this task. It is hoped a more exact account will be set forth for publication elsewhere, and that enough is said here to define all but the details of the main purpose of this paper—explaining ALGOL 60 in terms of IAEs.

THE SHARING MACHINE

In [MEE] we described an abstract machine, the SECD-machine, capable of evaluating an AE. Although the value of an AE was defined independently of this machine, it was given a more definite status by being backed up by a specific machine. The notion of the meaning of an IAE, on the other hand, is completely dependent on an abstract machine, the "sharing machine," for executing IAEs—the word "execute" seems more appropriate than "evaluate" in the case of IAEs. The sharing machine is an elaboration and an extension of the SECD-machine. The elaboration is concerned with modeling the fact that distinct state-positions having equal occupants might "share" the same representation and hence get updated collectively. The extension is concerned with the execution of the two new features, namely assigners and J (covered above).

The Transition Rule of the Sharing Machine. The main features of the elaboration are the following four specific rules governing whether or not two state-positions "share." For, each state of the sharing machine is characterized by an SECD-state *together with* an equivalence relation, namely "sharing," among its component-positions. In the straightforward computer representation, each equivalence class corresponds to an address. For each step we must say how this equivalence relation changes.

When an identifier is scanned, the stack-head is left sharing with the environment position holding the identifier's value. Also when a closure is applied, the youngest level of the new environment shares with any surviving "co-sharers" of the old stack-head (i.e. the argument). As a consequence of these two provisions a function can achieve nonlocal effects by assigning to its formals (i.e. arguments

[3] I think this term is due to Peter Naur.

are called by "simple name" [8]—rechristened "reference" in CPL [1]; this is the mode used for arguments of FORTRAN functions). They also ensure that no special provision is needed for scanning an identifier when it occurs as the lhs of an assigner. Since the classification of identifiers into constants and variables is not reflected in the behavior of the machine, constants are not invulnerable to resetting.

A third rule is that when a closure is applied, the components of older levels of the new environment share with corresponding components of the environment from which it was derived. As a consequence of this provision, a function can achieve nonlocal effects by assigning to its free identifiers (including to constants).

Fourthly, when a control string is exhausted the new stack-head is left sharing with any surviving co-sharers of the old stack-head. This provision ensures that an application of a function can be appropriate as a lhs, e.g.

$$\{\lambda x.\ if\ (x = 0)\ (x, b)\}[a] \Leftarrow \cdots$$
$$(\lambda().\ a_{i,j})() \Leftarrow \cdots$$

The Function "separate". There is a function *separate* that enables us to avoid nonlocal effects at will. That is to say, suppose the current stack-head has been obtained by loading some environment component (e.g. the value of an identifier of subscripted identifier); then if it is later incorporated in a new environment the above-mentioned provisions ensure that any assignment to its new position will also reset its old position. This is avoided if it is subjected to *separate* before incorporation in a new environment. (There are other suggestive names for *separate* that are rejected here since their suggestiveness does not entirely avoid misleading: *copy* is used in LISP [4] to mean copying all components "down to the level of" atoms, i.e. components named by variables (as opposed to constants), whereas we postulate that *separate* insulates *every* component from future resetting; **value** as used in ALGOL 60 coincides with *separate* in the case of operands that are suitable lhs's, but has acquired too many mutually incompatible connotations in discussing three relationships that we wish to delineate clearly, namely the relation between expressions and their denotations, between functions and their results, and between state-positions and their occupants.)

Executing an assigner. The *lhs* and *rhs* of an assigner can be evaluated in the same way as each other. The only difference is that, to be appropriate on the lhs, an IAE must denote some previously produced object, for example a named object or component of a named object. Every intermediate result of evaluation occupies a certain position in the current state of the machine; hence a lhs expression determines a state-position. Scanning an assigner resets this state-position and also every state-position sharing with it; it leaves a nugatory result on the stack, namely *nullist*.

The "meaning" of an IAE. In the AE/SECD system, each AE denotes an abstract object that is completely characterized by (a) the result produced by applying it to each abstract object that is amenable to it, and (b) the result yielded by subjecting it to each abstract object that is applicable to it.

With IAEs the situation can be very much more complicated. For each case of application the question arises, not merely what result is loaded onto the stack, but also, for each possible pattern of sharing throughout the current state, how it is changed. However, in the case of the primitives we use to model ALGOL 60, it is possible to overlook most of this complication. All but two are straightforward functions without side-effects. The two exceptions are *separate*, and *assignandhold*, which is a dressed-up version of the *assigner* format. For present purposes we can roughly say that the meaning of an IAE has two aspects, a descriptive and an imperative aspect, of which one or other may be unimportant. The descriptive aspect corresponds to the value, or denotation, of an AE. The imperative aspect corresponds to the change of machine state caused by executing the IAE.

ALGOL 60 AS SUGARED IAEs

The sharing machine provides a precise criterion for the correctness of the correspondence between ALGOL 60 and IAEs, namely that they should have corresponding effects when executed in corresponding environments. On account of the absence of specified input/output facilities in ALGOL 60, the meaning of this criterion is less clear with regard to whole programs than it is with regard to subblocks, operating on, and producing, the values of declared variables. However, input/output devices can be modeled as named lists, with special, rather restricted functions associated. Reading is modeled by a procedure (or function) that operates on a list, resets it by removing some initial segment, and also resets other variables with values derived from the initial segment (or, if a function, produces these values as its result). Writing is modeled by a procedure that operates on a list, and appends a new final segment derived from other variables. (Alternatively, a purely functional approach can be contrived by including the transformed list among the results.) So no new principle is raised by input/output, nor hence by whole programs.

We give later a formal presentation of a function that associates an IAE with each ALGOL 60 program. This function is intended to satisfy the criterion stated above. Discovering whether it does or not is a task that can be approached in two ways: either experimentally, using an implementation of the sharing machine; or with pen and paper, developing a proof. The length of what one accepts as a satisfactory proof will depend on his intuitive grasp of component ideas used in the formalization.

This formalization determines the "syntax" of ALGOL 60 in the special sense explained above. The identifiers occurring free in the IAEs that model ALGOL 60, will be the "constants" of ALGOL 60 in our special sense. They comprise the nine standard function identifiers, the score or so

of such symbols as $+$, $<$ and $\wedge$, the numerical, Boolean and character-string constants, a handful of functions to deal with ALGOL 60's array and iteration facilities, and another handful that are so unproblem-oriented that they are probably implicit in any tolerable language (in the sense that they would be needed were it subjected to the treatment that ALGOL 60 gets here).

By saying what each constant denotes, we shall be saying what are the "primitives" of ALGOL 60. The closure[4] under application and abstraction of this set of primitives is the "universe of discourse" of ALGOL 60. Actually, the syntax of ALGOL 60 is such that some IAEs have no written representation, and even such that some members of the universe of discourse (for example function-producing functions) are not denoted by any text.

Informal Presentation of the Correspondence

We now give a detailed but informal description of a correspondence between expressions of ALGOL 60 and IAEs. The interest of this particular correspondence is that it is "correct" in the sense put forward above; i.e. corresponding expressions executed in corresponding environments have corresponding outcomes. Subsequently we formalize the correspondence. However the informal treatment includes some material not covered by the formalization, namely a description of the "basic ALGOL 60 environment", i.e. the primitive objects whose names appear in the IAEs that model ALGOL 60. If our formalization were to not merely specify the correspondence but also prove its correctness, then it would have to include a formal specification of these primitives.

There is often considerable choice of IAEs to model a particular ALGOL 60 expression. This is true even when the primitives and the "constants" that name them have been chosen. In particular there is a conflict between using IAEs that correspond naturally in each individual case, and using a uniform and easily specified rule. For this reason the illustrations in our informal account sometimes deviate from the general rule presented in the formal account that follows it.

BRIEF OUTLINE

Table 1 gives a rough indication of the correspondence.

There are two features of ALGOL 60 that give rise to particularly clumsy IAEs.

1. Own identifiers declared other than in the head of the body of a globally declared procedure.
2. Conditional statements that are entered unnaturally (i.e. by a **go to**) and exited naturally (i.e. other than by a **go to**).

THE DOMAIN OF REFERENCE OF ALGOL 60

The correspondence given in this paper associates with each ALGOL 60 text an IAE whose principal significance is either denotational or state-transformational. If the former it denotes either an integer, a real (we postulate two disjunct classes of abstract objects—so 3 and 3.0 are not equivalent), a truth-value or a character-string; or a list, array or function. State-transformational IAEs model statements and labels; a switch is related to such IAEs in much the same way as a vector is related to numerical expressions.

Declarations are considered as giving initial values to the local identifiers. For instance integer and real identifiers are initialized respectively to integer zero and real zero. A Boolean is initialized to **false**. Switches and procedures are initialized to vectors and functions respectively (and not subsequently reset).

Lists are characterized by the following structure definition.

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

TABLE 1

ALGOL 60	*IAEs*
Identifiers, operator symbols, also some special words and configurations	Identifiers
Local identifiers	Variables bound in a λ-expression occurring as operator
Formal parameters	Variables bound in a λ-expression occurring as operand
Function designator, subscripted variable, and procedure statement	Operator/operand combination
Procedure	λ-expression
Actual parameter called by name	λ ( )-expression
Occurrence of a formal called by name	Application to null operand list
Value part of a procedure declaration	Auxiliary definition qualifying the procedure body, redefining some of the formals
Specification part	Auxiliary definition qualifying the procedure body, redefining some of the formals
Block	Combination whose operator is the block-body, and whose operand denotes the (possibly concocted) initial values of the locals
Statement	Expression denoting a none-adic function, changing the environment by side-effects
Compound statement	Functional product of none-adic functions
Label	Identifier defined by a none-adic program point
Labeled segment of program	Program point whose body, denotes a none-adic function
go to—statement	Last (i.e. outer) term of a functional product
Switch	Vector of program closures
Conditional expression or statement	Selection of an item from a listing

[4] In the usual algebraic sense, not the special sense attributed to this word by me here and in [MEE].

In our model every procedure, switch and array operates on a list of zero or more arguments. An expression that denotes a list in terms of subexpressions denoting each item of the list is called a *listing*. For example,

$(a+b,\ c+d,\ e+f)$
$(a+b,\ c+d)$
$unitlist(a+b)$ or $u(a+b)$
$()$

are listings. In [MEE] the first of these was considered as a convenient way of writing

$$prefix(a+b)(prefix(c+d)(prefix(e+f)()))$$

This, coupled with the fact that operands are evaluated before operators, ensures that the items of a listing are evaluated in the right-to-left order. Hence in transcribing from ALGOL 60 to IAEs the order of the items of every listing (including the implicit listings of operands of $+$, $=$, etc.) must be reversed. This minor complication is avoided here by adopting a different analysis of listings, by which

$$(a+b,\ c+d,\ e+f)$$

is considered as a convenient way of writing

$$suffix(e+f)(suffix(c+d)(suffix(a+b)()))$$

(It might alternatively have been avoided by varying the evaluation mechanism so as to evaluate operators *before* operands. In view of the limited form of operator occurring in ALGOL 60 such a change would have few other repercussions.)

There is no feature of the correspondence that "explains" the left-to-right rule of ALGOL 60. In this respect the correctness of the model depends on a similar rule for IAEs. We can put this another way. The correspondence presented in this paper "explains" semantic features of ALGOL 60 in terms of syntactic, or more precisely structural, features of IAEs. But the left-to-right rule is a semantic feature of ALGOL 60 that relies for its explanation on a *semantic* feature of IAEs. The semantic feature we use is that operands are evaluated before operators. (A logically more economical approach would use merely the fact that an operand to a λ-expression is evaluated before its λ-body. Thus in evaluating $(\lambda x.a^2+x^2)(b+c)$, the subexpression $b+c$ is evaluated before a^2, whether the machine evaluates an operand before, after or concurrently with its operator.)

An array is considered as a function whose domain is a subset of the set of integer-lists. It is initialized with the appropriate domain (not subsequently altered) and with all its elements equal. Thus

$$expandtoarray((0,\ m),\ (0,\ n))(a)$$

denotes an $(m+1) \times (n+1)$ array each of whose elements is a. An own array is initialized with the appropriate dimensionality but with array bounds $(-\infty,\ +\infty)$. These are "pared down" to finite values at the first entry to the array's block. Thus if A is a two-dimensional array then

$$parearray((0,\ m),\ (0,\ n))(a,\ A)$$

denotes an $(m+1) \times (n+1)$ array whose elements are the same as those of A insofar as their domains overlap, and otherwise a.

A switch is initialized by the function *arrangeasarray*, that transforms a given list structure into an array of given domain, e.g.

$$arrangeasarray((0,\ 2),\ (0,\ 3))((a,\ b,\ c,\ d),\ (e,\ f,\ g,\ h),\ (i,\ j,\ k,\ l))$$

denotes a 3×4 array whose elements (row by row) are $a, b, c, \cdots, k, l$.

FOR-LISTS

Let us use the term "control-list" to mean the list of successive values assigned to the controlled variable during one execution of a **for**-statement. The point of departure of our treatment of **for**-statements is that a **for**-list might roughly be said to "denote" the control-list, with each **for**-list-element denoting one segment of it. This suggests the following incorrect rendering.

```
for v := a step b until c,        for(v,
         d,                         concatenate (step(a, b, c),
         e while p                    unitlist (d),
  do T                                while(e, p)),
                                    T)
```

where *for*, *concatenate*, *step* and *while* are defined as follows.[5]

```
rec for(v, S, T)    = if ¬ null S then [v := hS;
                                        T;
                                        for(v, tS, T)]
rec concatenate S   = null S → ( )
                      null(hS) → concatenate (tS)
                      else → h²S:concatenate(t(hS):tS)
rec step(a, b, c)   = (a − c) × sign(b) > 0 → ( )
                      else → a:step(a+b, b, c)
rec while(e, p)     = p → e:while(e, p)
                      else → ( )
```

However, these definitions fail to reflect the sequence of execution prescribed for ALGOL 60. When interpreted by the sharing machine they would lead to an attempt to evaluate the entire control-list before the first iteration of the loop. The inadequacy of this approach is especially flagrant in the case of *while*. We therefore consider **for**-list-elements as denoting not lists but a particular kind of function, called here a *stream*, that is like a list but has special properties related to the sequencing of evaluation. Principally, the items of an intermediately resulting stream need never exist simultaneously. So streams might have practical advantages when a list is subjected to a cascade of editing processes.[6]

---

[5] Following [MEE], an infixed colon indicates prefixing. Thus "x:L" is equivalent to "*prefix x L*."

[6] It appears that in stream-transformers we have a functional analogue of what Conway [12] calls "co-routines."

However, the user of a purely functional system (i.e. AE/SECD rather than IAE/sharing machine) would have no way of telling whether his intermediately resulting lists were in fact being streamed or not, since the only differences in outcome are concerned with the amount of store used, or the range of jobs possible with a given size of store. On the other hand, the introduction of imperatives makes it possible to write list-expressions whose outcome *is* affected by whether they are represented as streams or not. Hence it becomes necessary to introduce a new set of identifiers that play the same role for streams that *h*, *t*, etc. play for lists. The next subsection is concerned with these operations.

Streams

There is a relationship between lists and functions that is used here in modeling **for**-statements (and would be used to model input/output if Algol 60 included such). In this relationship a nonnull list L is mirrored by a none-adic function S that produces a 2-list consisting of (1) the head of L, and (2) the function mirroring the tail of L. The common functions, etc. associated with lists are mirrored as follows.

nullist	$\lambda(\,).(\,)$
null(*L*)	*null*(*S*( ))
head(*L*)	*1st*(*S*( ))
tail(*L*)	*2nd*(*S*( ))
prefix(*x*)(*L*)	$\lambda(\,).(x, S)$
cons(*x*, *L*)	$\lambda(\,).(x, S)$
unitlist(*x*)	$\lambda(\,).(x,\lambda(\,).(\,))$

It is easy to see that the first five expressions on the right satisfy the four relationships that characterize *nullist*, *null*, *h*, *t* and *prefix*.

null(*nullist*)	*null*({$\lambda(\,).(\,)$}[ ])
$\neg$*null*(*prefix xL*)	$\neg$*null*({$\lambda(\,).(x,S)$}[ ])
h(*prefix xL*) = *x*	*1st*({$\lambda(\,).(x,S)$}[ ]) = *x*
t(*prefix xL*) = *L*	*2nd*({$\lambda(\,).(x,S)$}[ ]) = *S*

This correspondence serves two related purposes. It enables us to perform operations on lists (such as generating them, mapping them, concatenating them) without using an "extensive," item-by-item representation of the intermediately resulting lists; and it enables us to postpone the evaluation of the expressions specifying the items of a list until they are actually needed. The second of these is what interests us here.

The expressions that make use of this technique can be made slightly clearer by using the following definitions.

*nullist** = $\lambda(\,).(\,)$
*null**(*S*) = *null*(*S*( ))
*h**(*S*) = *1st*(*S*( ))
*t**(*S*) = *2nd*(*S*( ))

However, the analogous definitions for the constructors cannot be used since they would not preserve the sequencing of evaluation. The best that can be done is to introduce a new *syntactic device* whereby for any two expressions L, M

$$L{:}{*}M \quad \text{stands for} \quad \lambda(\,).(L, M)$$

We now define functions that correctly mirror Algol 60's three kinds of **for**-list-element.

rec *step**(*a*, *b*, *c*) =
  $\lambda(\,).[(a'-c') \times sign(b') > 0] \rightarrow (\,)$
    else $\rightarrow$ [*a'*, *step**($\lambda(\,).a'+b'$, *b*, *c*)]
    where *a'*, *b'*, *c'* = *a*( ),*b*( ),*c*( )
*unitlist**(*a*) = *a*( ) :* *nullist**
rec *while**(*e*, *p*) = $\lambda(\,).p' \rightarrow$ [*e'*, *while**(*e*, *p*)]
    else $\rightarrow$ ( )
    where *e'*, *p'* = *e*( ), *p*( )

The matching definitions for *concatenate** and *for** should be obvious.

The above formulas reflect certain choices of Algol 60's designers, e.g.

(a) that all parameters are evaluated "when they are come to," rather than e.g. evaluating the parameters of a **step**-element (arithmetic progression) all together;

(b) that the decision whether the current iteration is the last is taken *after* it, not before it;

(c) that any resetting of the controlled variable during the execution of the **for**-body affects its subsequent values.

That is to say, had different choices been made in these matters, then a different IAE, or different definitions of the auxiliary functions involved, would have been needed to mirror **for**-statements.

Types

Roughly speaking our model deals with types "interpretively." Specifiers in Algol 60 affect the prescribed outcome of a program only by causing transfer between real and integer, or by rejecting an argument outside the specified class. We suppose that associated with each specifiable class there is a transfer function whose range is within that class. For instance, *float* is defined for numbers, i.e. for reals and integers, and leaves reals unchanged; similarly with *unfloat*, which is prescribed in the Algol 60 report to be defined by

$$unfloat(x) = entier(x + 0.5)$$

The transfer function for truthvalues is merely a very limited form of the identity function that is defined by

$$rejectallbuttruthvalues(x) = \textbf{Boolean}(x) \rightarrow x$$

(Here **Boolean** designates the class—or predicate, we do not distinguish—whose members are the truthvalues. A conditional expression none of whose conditions hold is taken as undefined.) More generally we define a function *in* as follows:

$$in(A)(x) = A(x) \rightarrow x$$

so that if A is a class then inA is a filter that rejects nonmembers of A. For example in(**Boolean**) is the function *rejectallbuttruthvalues* defined above. So the transfer functions for Booleans and strings are respectively in(**Boolean**) and in(**string**).

The function defined by

$$floatresult(f)(x) = float(f(x))$$

transforms any number-producing function into a real-producing function. (This definition exploits the fact that we consider any function as operating on a *single* argument allbeit a list.) More generally, if t is the transfer function for some class A, then the transfer function for A-producers is Bt, where B (Curry's combinator **B** [2]) is defined by

$$Btfx = t(fx)$$

So for instance *Bfloat* is the function *floatresult* defined above. The transfer functions for type-procedures are therefore *Bfloat*, *Bunfloat* and $B(in(\textbf{Boolean}))$. Since arrays are treated as functions these also serve as transfer functions for arrays.

Imperatives are treated as nullist-producing functions; so it would appear that the best we can do for a transfer function for labels is $B(in(null))$. Hence the transfer functions for nontype procedures and for switches (which are considered as arrays whose elements are program-closures) is $B(B(in(null)))$.

The effect of the above provisions for checking arguments is that a mismatched procedure, array, label or switch is not itself immediately rejected. Instead it is modified so that any result it produces, whenever and if ever it is applied, is rejected. Hence our model is overtolerant in that a mismatch will not lead to rejection if the procedure is never applied, or if it is exited unnaturally and thus evades producing a result. Furthermore, a label denotes a program-closure and so even when its result, namely *nullist*, is produced, the context is never resumed and so the check never occurs. Hence the identity function serves equally well as transfer function.

The Constants and Primitives of Algol 60

The correspondence given in this paper associates with each Algol 60 text an IAE in which the identifiers occurring free are drawn from the following three groups.

Group 1 consists of the arithmetical, Boolean and string constants. An arithmetical constant is an unsigned number as defined in the Algol 60 report, and designates an integer or a (rational) real. However the integers also include '$-\infty$' and '$+\infty$', used in the initial array bounds of own arrays. The Boolean constants are **'true'** and **'false'** and designate the truthvalues. The string constants are certain character-strings whose first and last items are ' ⌜ ' and ' ⌝ ', respectively; such a constant designates the string obtained by removing its first and last items. (It would be possible to avoid an infinity of primitives by considering each written number and each character-string as having internal applicative structure. These might conveniently use such number-forming and string-forming functions as: $decimal(m,n) = 10m + n$; $quote(s) = concatenate(u' \ulcorner\ ',\ s,\ u' \urcorner\ ')$.)

Group 2 consists of symbols and identifiers whose meaning is laid down in the Algol 60 report, and also a number of identifiers coined by us and explained above. We assume that any collisions between these coinages and Algol 60 identifiers are avoided by some device such as the use of a different typeface, e.g. italic instead of roman.

$+, -, \times, /, \div, \uparrow, +_M, -_M, <, \leqq, =, \geqq, >, \neq, \neg, \wedge, \vee, \supset, \equiv,$

abs, sign, sqrt, sin, cos, arctan, ln, exp, entier

The infixed operators are taken as applying to 2-lists, either 2-number-lists or 2-Boolean-lists. Numerical functions are applicable to both reals, and integers; if n and $float(n)$ are both amenable to a function then they yield the same result. The coinages are

for, concatenate, step*, unitlist*, while*, expandtoarray, arrangeasarray, parearray, float, unfloat, in,* **real, integer, Boolean, string,** *atom*

Group 3 consists of names for very basic objects.

null, nullist, suffix, if, B, K, I, Y, separate, assignandhold

null is the predicate that tests whether a list has zero length.

nullist is a list of length zero.

suffix makes a list one longer, e.g.

$suffix(x)(a,b,c) = (a,b,c,x)$

if satisfies the following:

$if(\textbf{true}) = 1st$

$if(\textbf{false}) = 2nd$

B forms a functional product

$B(t)(f) = \lambda x.t(f(x))$

It is used in delaying transfer functions for type procedures and formals called by name.

K produces "constant functions"

$K(x)(y) = x$

So for instance $K3$ is a function whose result is 3 for any argument; it is used to tidy up assignments.

I is the identity function, defined by

$I(x) = x$

It plays the role of dummy statements.

Y is the "fixed-point finder." In so far as it is reasonably representable it can be defined by

$$Y(F) = \textbf{let}\ z = separate(nullist)\ \textbf{let}\ z' = F(z)\ 2nd((z \Leftarrow z'),z)$$

This definition relies on the fact that when a function-transformer is applied to the (arbitrarily chosen) argument *nullist*, rejection does not occur unless, or until, the argument is actually applied.

separate avoids unwanted side-effects; it is used when parameters are called by value.

assignandhold is defined by

$$assignandhold(x)(y) = \textbf{let}\ x = \textbf{real}\ y \rightarrow float\ x,\ \textbf{integer}\ y \rightarrow unfloat\ x,\ \textbf{Boolean}\ y \rightarrow in(\textbf{Boolean})x\ 2nd((y \Leftarrow x),x)$$

In this subsection and the four preceding ones we have characterized the abstract objects comprising the "domain of reference" that our analysis imputes to Algol 60. The characterization has been partly formal and partly informal, taking for granted such things as numbers, propositional relations, etc.[7] In the next subsection we turn to the main topic of this paper, namely how Algol 60 texts can be construed as IAEs referring to these abstract objects.

[7] In [9], Böhm is concerned with the formal treatment of this topic.

Illustrations of the Correspondence

Each example below illustrates the correspondence between a particular feature of Algol 60 and a particular feature of IAEs. That is to say, each example illustrates a rule for eliminating a particular feature of Algol 60 in terms of IAEs. In order to turn a piece of Algol 60 into an IAE it will usually be necessary to make many successive applications of these rules. In some of the examples the transformation into an IAE has been only partially performed to better emphasize the particular point being made by the example. So the right-hand half of an illustration may not always contain an IAE, but it always contains something that is nearer to one of the four forms of IAE than the left-hand, Algol 60 expression.

Identifiers. Except for the treatment of individual occurrences of identifiers, the correspondence is context independent. The IAE corresponding to each occurrence of an Algol 60 identifier depends on the way the identifier is declared or otherwise introduced, as follows.

1. Whenever a local declared to identify a type procedure occurs as a left-hand side within its declaration, it is replaced by a variant identifier, indicated here by decorating it with an asterisk. We call this the *result variant* of the procedure identifier.

2. Every occurrence of a formal not specified by value, or of a local declared to identify a parameterless procedure, is (provided 1 does not apply) modified by attaching an empty operand listing to it. This is indicated here by an empty bracket pair. Thus we treat actual parameters called by name, and parameterless procedures, as none-adic functions.

3. Owns are treated as globally declared and are replaced by variants to avoid collision between two declarations of the same identifier. These variants are called *own variants* of the identifier appearing in the text and are indicated here by decorating it with one or more daggers. (This is one of the two unsatisfactory features of the correspondence between Algol 60 and IAEs. Which party is to blame is a question we return to later.)

Here is an example of these substitutions.

```
begin                                  begin
  own integer a;                         own integer a†;
  real procedure f(x,y,g);               own real a††;
    value x;  real x;                    real procedure f(x,y,g);
    begin                                  value x,y,g;
      own real a;                          begin
      y:=x+g(y)+a;                           y( ):=x+g( )(y( ))+a††;
      f:=k(f,g,f(x,y,j))                     f*:=k(f,g( ),f(x,y( ),j))
    end                                    end
  real procedure h;                      real procedure h( );
    h:=k(f,j,h);                           h*:=k(f,j,h( ));
  a:=f(a,h,f)                            a†:=f(a†,h( ),f)
end                                    end
```

With the exception of these provisions each subsequent example is self-contained; that is to say the given bit of Algol 60 corresponds to the given IAE whatever its context (provided it is grouped sensibly—for instance, not every occurrence of the character-string '$a+b$' in Algol 60 corresponds to the IAE "$+(a, b)$"—witness '$c\times a+b\times d$').

Variables. The treatment of individual identifiers has already been explained. In summary

x	formal called by name	$x( )$
	type procedure as *lhs*	x^*
	parameterless procedure	$x( )$
	own identifier	$x^\dagger, x^{\dagger\dagger}$, etc.
	straightforward	x

If all the identifiers involved are straightforward the treatment of subscripted identifiers and function designators is as follows.

$A[i+j, k]$ $\quad$ $A(i+j, k)$

$f(x+y, z)$ $\quad$ $f(\lambda( ).x+y, \lambda( ).z)$

The transformation of actual parameters into none-adic functions is associated with the possibility that a procedure might call its formals by name. It is complemented by a peculiarity in the treatment of procedure declarations as noted below.

If the identifiers involved are all formals called by name then the transcription to IAEs is

$$A( )(i( ) + j( ), k( ))$$

$$f( )(\lambda( ).x( ) + y( ), \lambda( ).z( ))$$

In the last example, '$\lambda( ).z( )$' can be replaced by 'z'. In future illustrations all occurrences of identifiers are assumed to be straightforward unless the reason for nonstraightforwardness is contained in the example. Any nonstraightforwardness would involve superimposing the appropriate treatment on any other transformations that are needed.

Expressions.

$a + b$ $\quad$ $+(suffix\ b\ (suffix\ a\ (\ )))$

In future illustrations (as in previous ones) the treatment of argument lists, as also usually of infixed operators, will be taken for granted.

$-a + b$ $\quad$ $+(-_M(a), b)$

The symbol $-_M$ designates the monadic function "negate."

$a+b-c+d$ $\quad$ $+(-(+(a,b),c),d)$

This example shows how "left association" is reflected (as opposed to the left-to-right rule which is quite independent). The following examples show how the "precedence" rules are reflected.

$c\times a+b\times d$ $\quad$ $+(\times(c,a), \times(b,d))$

$a-b/c\uparrow d$ $\quad$ $-(a,/(b,\uparrow(c,d)))$

$a\uparrow b/c-d$ $\quad$ $-(/(\uparrow(a,b), c), d)$

$p\wedge q\vee r\wedge s$ $\quad$ $\vee(\wedge(p,q), \wedge(r,s))$

Conditional expressions use the function *if*, such that

if **true** = 1*st*

if **false** = 2*nd*

E.g.

if p **then** a **else** b	$(if\ p)(\lambda(\ ).a, \lambda(\ ).b)(\ )$

So the evaluation of the above IAE involves selecting one from a list of two none-adic functions and then applying the selected one to the nullist.

Blocks. Each declaration is construed as a definition whose definee consists of one or more local identifiers and whose definiens denotes initial values for them. In the case of type and array declarations the initial value is concocted with zeros. In the case of switches and procedures the definiens is already in ALGOL 60. (In fact the initial value is never changed since there are no assignments to switch and procedure identifiers.)

Array declarations are initialized in terms of nonlocals (e.g. a procedure identifier declared in parallel with an array cannot be used in the array bound expressions). On the other hand switches and procedures may refer to locals, i.e. to arrays, reals, etc., and also to each other. Hence array definitions qualify switch and procedure definitions (as well as the block body). Whereas switch and procedure definitions do not qualify array definitions, they do qualify themselves; that is to say they may be "self-referential," i.e. "circular" or "recursive." We accordingly speak of "recursive" and "nonrecursive" definitions or declarations. This dichotomy leaves type declarations uncommitted since they contain no initializing expression. If they did it would be difficult to mechanize self-referential initializations (e.g. **rec** $x = x^2 + \frac{1}{4}$); so we class them as nonrecursive.

Definitions can also arise from the block-body—their definees being the labels that are local to the block. These are defined in terms of locals, including each other, and they may be referred to by procedure and switch declarations. Hence labels must be grouped with switches and procedures as a single simultaneously recursive definition. The overall treatment of a block is therefore as follows.

begin real a;	**let** $a = \phi_1$
array A ϕ_2 ;	**and** $A = \phi_2$
procedure P ϕ_3 ;	**let rec** $P = \phi_3$
switch S ϕ_4 ;	**and** $S = \phi_4$
ϕ_5 ;	**and** $L = \phi_6$
L: ϕ_6 ;	**and** $M = \phi_7$
M: ϕ_7	ϕ_5
end	

i.e.

$$\{\lambda(a,A).\{\lambda(P,S,L,M).\phi_5\}$$
$$[Y\lambda(P,S,L,M).(\phi_3,\phi_4,\phi_6,\phi_7)]\}$$
$$[\phi_1,\phi_2]$$

The detailed structure of the ϕ's is the subject of the following sections. In the last example, and in some that follow, the IAE was presented twice, in a less and more formal notation. This emphasizes the fact that the correspondence being illustrated is between ALGOL 60 texts and certain *abstract* objects, not written representations of them.

Pseudoblocks. There are three contexts that may or may not be occupied by blocks, but are in any case like blocks in the treatment of labels. These contexts are procedure-bodies, **for**-bodies and whole programs. We call them *pseudoblocks.* Whole programs must also be qualified by a conglomeration of all the own-declarations that occur in the program. Their role in a pseudoblock is the same as that of the nonrecursive declarations in a block.

Declarations.

real x	**let** x = *separate* (0.0)
integer i, j	**let** i = *separate* (0)
	and j = *separate* (0)
Boolean p	**let** p = *separate* (**false**)

By initializing every local identifier, IAEs impute meaning to certain ALGOL 60-like programs to which the ALGOL 60 report prescribes no meaning. The use of *separate* prevents subsequent assignments from altering the environmental objects designated by '0.0', '0' and **'false'**.

real array $a[i:j, k:l]$	**let** a = *expandtoarray*$((i,j),(k,l))$ (0.0)
integer array $a, b, c[i:j]$	**let** (a,b,c) = ($separate\ A$, $separate\ A$, $separate\ A$) **where** A = *expandtoarray* $(u(i,j))(0)$
array $a, b, c[1:n], d[1:r,1:s]$	**let** $((a,b,c),u(d))$ = (($separate\ A$, $separate\ A$, $separate\ A$) **where** A = *expandtoarray* $(u(1,n))(x)$, *unitlist* (*separate* A) **where** A = *expandtoarray* $((1,r),(1,s))$ (x)) **where** x = 0.0
switch $S := L, M, N$	**let** S = *arrangeasarray* $(u(1,3))$ (L,M,N)

A procedure declaration is treated as a definition whose definiens is a λ-expression. Our treatment of actual parameters matches that of formals called by name. So if a formal is called by value the procedure body needs some decoration.

procedure $f(x,y,z)$;	**let** $f(x,y,z)$ = **let** x = *separate*$(x(\ ))$
value x, z;	**and** z = *separate*$(z(\ ))$
ϕ	ϕ
i.e.	**let** $f = \lambda(x,y,z).$ $\{\lambda(x,z).\phi\}$ [*separate*$(x(\ ))$, *separate*$(z(\ ))$]

If *separate* were omitted the effect would be that of "calling by simple name."

Specifiers also involve decorating the body.

procedure $f(x,r,p)$;	**let** $f(x,r,p)$ =
value x, r, p;	**let** x = *separate*$(x(\ ))$
real x;	**and** r = *separate*$(r(\ ))$
integer r;	**and** p = *separate*$(p(\ ))$
Boolean p;	**let** x = *float*(x)
$S(x+r, p)$	**and** r = *unfloat*(r)
	and p = *in*(**Boolean**)(p)
	$S(\lambda(\ ).x+r, \lambda(\ ).p)$

The specification of a formal called by name involves *preparing* to transfer its result.

procedure $f(x,r,s,P)$;	**let** $f(x,r,s,P)$ =
value x;	**let** x = $separate(x(\,))$
real x;	**let** x = $float(x)$
integer r;	**and** r = $Bunfloat(r)$
string s;	**and** s = $B(in\ \mathbf{string})(s)$
real procedure P;	**and** P = $B(Bfloat)(P)$
$p(x+r,\ s)$	$P(\,)(\lambda(\,).x+r(\,),\ \lambda(\,).s(\,))$

This treatment of procedures as arguments might be clarified by observing that it suggests a particular extension of ALGOL 60, by which, for example

if $a<b$ **then** *sin* **else** *cos*

would be a permissible actual parameter. There would then be a natural sense in which one might distinguish calling procedures by name or value.

This extension raises a special question concerning *parameterless* procedures, since at first sight the designers of ALGOL 60 appear to have forestalled it by giving an incompatible meaning to calling a parameterless procedure by value. However, the situation is saved by other limitations in ALGOL 60. For, it uses the identifier of a parameterless procedure, say 'p', as an abbreviation for its result, i.e. for '$p(\,)$'. It is thus impossible in ALGOL 60 to refer to a parameterless procedure except in the context of applying it. Thus all "genuine" cases of procedures as arguments are called by name. And, as it happens, for all such operands that ALGOL 60 allows (namely identifiers) calling by name and calling by value have the same outcome, since procedure identifiers cannot be assigned to in ALGOL 60.

A parameterless procedure gives rise to an IAE with an *explicit* null operand listing, as in

procedure p; $a := b$ **let** $p = \lambda(\,).(a := b)$

The next example shows how a type procedure can be matched by an IAE.

real procedure $f(y)$	**let** $f(y)$ = $2nd\ [(f^* := ay^2+by+c),$
value y;	$f^*]$
$f := a\times y\uparrow 2 + b\times y + c$	**where** y = $separate(y(\,))$
	where f^* = 0.0

Notice the way in which the IAE distinguishes the two roles played by the type procedure identifier. It is illuminating to compare the above expression with the following equivalent:

let $f(y) = ay^2 + by + c$

There are two provisions needed in general for rendering ALGOL 60 that are otiose in this particular example: (i) the ability to *re*assign to the result variable and to include assignments to other variables, and (ii) the ability to call parameters by name.

Statements. Each statement is rendered as a 0-list-transformer, i.e. a none-adic function producing the nullist for its result. It achieves by side-effects a transformation of the current state of evaluation.

Since the execution of the IAE *assigner* format makes no provision for type transfer, assignment statements are rendered in terms of *assignandhold*.

$a := b+c$	$\lambda(\,).K(\,)(assignandhold(b+c)(a))$
$a[i,j,k] := b+c$	$\lambda(\,).K(\,)(assignandhold(b+c)(a(i,j,k)))$

The operator $K(\,)$ is needed to ensure that an assignment *produces nullist*.

$a := b := c := d+e$	$\lambda(\,).a :=$
	$assignandhold(assignandhold(d+e)c)b$

Compound statements are considered as functional products (which we indicate informally by infixed dots).

begin R; S; T **end**	$\lambda(\,).T(S(R(\,)))$
i.e.	$T\cdot S\cdot R$

It should be observed that the dot notation, e.g. in $T(x)\cdot S(u, v, w)\cdot R$, is used here as an abbreviation for a $\lambda(\,)$-expression, and not for

$$B(B(Tx)(S(u, v, w))(R) \tag{1}$$

For, while (1) and

$$\lambda(\,).T(x)(S(u, v, w)(R(\,))) \tag{2}$$

are equivalent AEs, they are not equivalent IAEs. In fact the execution of (1) involves the execution of $S(u, v, w)$ and $T(x)$, whereas in (2) they will not be executed until (if ever) the resulting function is applied.

Dummy statements are construed as compounds containing no items.

begin end	$\lambda(\,).(\,)$

For-statements involve several auxiliary definitions, already explained.

for $v := a$ **step** b **until** c,	for$(v,$
$d,$	$concatenate^*$
e **while** p	$(step^*(\lambda(\,).a,\ \lambda(\,).b,\ \lambda(\,).c),$
do S	$unitlist^*(\lambda(\,).d),$
	$while^*(\lambda(\,).e,\ \lambda(\,).p)),$
	$S)$

Procedure statements are treated as function designators occurring as terms in the functional product.

$S;\ P(x,y);\ T$	$T\cdot P(x,y)\cdot S$

In a conditional statement the treatment for conditional expressions is superimposed on that for statements.

if p **then begin** P; Q **end**	if$(p)(\lambda(\,).\lambda(\,).Q(P(\,)),$
else begin R; S **end**	$\lambda(\,).\lambda(\,).S(R(\,)))$
	$(\,)$

Here some abbreviation is possible, namely

if$(p)(\lambda(\,).Q(P(\,)),$
$\lambda(\,).S(R(\,)))$

But this is not always so. For example in

if p **then** $P(x,y)$	if$(p)(\lambda(\,).P(\lambda(\,).x,\ \lambda(\,).y),$
else $R(u,v)$	$\lambda(\,).R(\lambda(\,).u,\ \lambda(\,).v))$
	$(\,)$

the corresponding abbreviation would result in both arms being executed.

One-armed conditionals are filled out with dummy second arms:

if p **then begin** P; Q **end** $\quad$ if$(p)(\lambda(\,).Q(P(\,)), I)$

Labels and Jumps. The treatment of jumps springs from the observation that the symbol **'go to'** in ALGOL 60 is redundant, and could be ignored by a processor. That is to say, there is a considerable similarity between labels and the identifiers of parameterless nontype procedures. It is possible to use the same "calling mechanism" for both, leaving any differences to be made by the thing that is "called." Thus there is a natural meaning to be given to a program that, at different times, substitutes labels and procedures for the same formal, e.g.

procedure P;
  if p **then go to** M;
...
L:...
... $f(P)$ ... $f(L)$...

It might therefore be supposed that labels can be eliminated formally by considering each labelled segment of program as a parameterless procedure declaration (and hence as a definition whose definiens is a $\lambda(\,)$-expression).

The present purpose is semantic specification, not cheap running. So this device is not invalidated by the fact that it involves accumulating a pile of "resumption points," one for every executed jump, that are never taken up. However, the device only yields a valid treatment of procedure exits at the cost of abandoning the facility for closed subroutines that is embodied in ALGOL 60's procedures. We are thus led[8] to "program-points." Labels are eliminated in favour of program-point declarations.

As mentioned earlier, a further complication is presented by the possibility that a statement can be entered unnaturally and then exited naturally. This is met by the formal treatment below.

Own Identifiers. The treatment of **own**'s can best be considered as involving a preliminary transformation of ALGOL 60, which eliminates **own** declarations except in the head of a whole program or the head of the body of a globally declared procedure. This transformation may require systematic changes of identifiers to avoid collisions.

The treatment of **own**'s would have been more elegant had they been associated with procedures instead of with blocks, and had their active life been prescribed as coterminous with the life of their procedure (which may include zero of more *activations* of the procedure). One consequence of this would have been that two nesting activations of a procedure would share their **own**'s or not, according as they were activations of the same life of the procedure or of two nesting lives. This meaning of **own** almost exactly coincides with the generally accepted meaning, if **own**'s are restricted to the heads of the bodies of procedures declared in the head of the whole program.[9]

Conclusion

One little sung use of ALGOL 60 has been as a standard with which to describe and compare other languages. Its suitability for this role arose from being described with remarkable precision, and from its greater power and elegance, so that its own idiosyncracies and limitations did not overshadow those of the languages being measured against it.

The language of IAEs is put forward here for consideration as a further step in this direction, and the supporting evidence is a detailed mapping of ALGOL 60 into IAEs. So far in this paper we have laid the groundwork for the mapping, namely a description of the "primitive objects" it refers to, and we have given specific instances of its application. The remainder of the paper is devoted to its formal characterization.

REFERENCES

1. BARRON, D. W., BUXTON, J. N., HARTLEY, D. F., NIXON, E., AND STRACHEY, C. The main features of CPL. *Comput. J. 6*, 2 (July 1963), 134–143.
2. CURRY, H. B., AND FEYS, R. *Combinatory Logic, Vol. 1.* North Holland, Amsterdam, 1958.
3. LANDIN, P. J. The mechanical evaluation of expressions. *Comput. J. 6*, 4 (Jan. 1964), 308–320.
4. MCCARTHY, J., ET AL. LISP 1.5 Programmer's Manual. MIT, Cambridge, 1962.
5. ——. Towards a mathematical science of computation. IFIP Munich Conference 1962. North Holland, Amsterdam, 1963.
6. NAUR, P., ET AL. Revised Report on the Algorithmic Language ALGOL 60. *Comm. ACM 6*, 1 (Jan. 1963), 1–17.
7. ROSENBLOOM, P. *The Elements of Mathematical Logic.* Dover, New York, 1950.
8. STRACHEY, C., AND WILKES, M. V. Some proposals for improving the efficiency of ALGOL 60. *Comm. ACM 4*, 11 (Nov. 1961), 488–491.
9. BÖHM, C. The CUCH as a formal and descriptive language. Presented at IFIP Working Conf., Baden, Sept. 1964.
10. LANDIN, P. J. A formal description of ALGOL 60. Presented at IFIP Working Conf., Baden, Sept. 1964.
11. VAN WIJNGAARDEN, A. Recursive definition of syntax and semantics. Presented at IFIP Working Conf., Baden, Sept. 1964.
12. CONWAY, M. E. Design of a separable transition-diagram compiler. *Comm. ACM 6*, 7 (July 1963), 396–408.

[8] In [11] van Wijngaarden meets this point by using an explicit "link" as an extra parameter to each procedure.

[9] For fuller discussion of the present approach to labels and **own**'s see [10].