5-1-1970

A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept

John C. Reynolds
Carnegie Mellon University, jr4g@andrew.cmu.edu

GEDANKEN–A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept

JOHN C. REYNOLDS
Argonne National Laboratory, Argonne, Illinois*

GEDANKEN is an experimental programming language with the following characteristics. (1) Any value which is permitted in some context of the language is permissible in any other meaningful context. In particular, functions and labels are permissible results of functions and values of variables. (2) Assignment and indirect addressing are formalized by introducing values, called references, which in turn possess other values. The assignment operation always affects the relation between some reference and its value. (3) All compound data structures are treated as functions. (4) Type declarations are not permitted.

The functional approach to data structures and the use of references insure that any process which accepts some data structure will accept any logically equivalent structure, regardless of its internal representation. More generally, any data structure may be implicit; i.e. it may be specified by giving an arbitrary algorithm for computing or accessing its components. The existence of label variables permits the construction of coroutines, quasi-parallel processes, and other unorthodox control mechanisms.

A variety of programming examples illustrates the generality of the language. Limitations and possible extensions are discussed briefly.

KEY WORDS AND PHRASES: programming language, data structure, reference, assignment, coroutine, quasi-parallel process, typeless language, applicative language, lambda calculus, list processing, nondeterministic algorithm
CR CATEGORIES: 4.20, 4.22, 5.23, 5.24

Introduction

The recent development of programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved problem. This paper describes an experimental language, called GEDANKEN, which was developed to attack this problem.

GEDANKEN is not intended to be a generally useful language, although it could be effective in situations where a fair degree of object program inefficiency is tolerable. Its major purpose (reflected in its name, which is meant as an analogy to gedankenexperiments in physics) is to explore the consequences of two basic design principles:

(1) *Completeness.* Any value which is permitted in some context of the language is permissible in any other meaningful context. In particular, functions and labels are permitted to be results of functions or values of references (e.g. variables), without imposing restrictions which maintain a stack discipline for run-time storage allocation.

(2) *The Reference Concept.* Assignment and indirect addressing are formalized in the following manner: among the possible values which may occur in a program are objects called *references*, which in turn possess other values. The assignment operation always affects the relation between some reference and its value.

Neither of these principles is novel. LISP [1a and 1b] (in its interpretive implementations), ISWIM [2], and PAL [3] all satisfy the principle of completeness, and the reference concept is used in ALGOL 68 [4] and BASEL [5]. But GEDANKEN goes beyond these languages in exploiting the power of these principles, i.e. in eliminating other language features which are rendered redundant by completeness and references. Specifically:

(1) The existence of function-returning and reference-returning functions allows all compound data structures to be treated as functions. For example, a one-dimensional ALGOL-like array is treated as a function whose domain is a finite set of consecutive integers and which maps each of these integers into a unique reference. This approach insures that any process which accepts some data structure will accept any logically equivalent structure, regardless of its internal representation. More generally, any data structure may be *implicit*; i.e. it may be specified by giving an arbitrary algorithm for computing or accessing its components. (Functional data structures have been suggested by Balzer [6], but his realization of the concept is quite different than GEDANKEN.)

(2) The existence of label variables permits the construction of coroutines, quasi-parallel processes, and other unorthodox control mechanisms. This is a direct consequence of not imposing a stack discipline on the program control information.

The main limitation of GEDANKEN is that declarations are not allowed to restrict the value ranges of identifiers, references, or function results. Languages with this property are usually called "typeless," although the types of values may be tested during execution. We do not suggest that type declarations are unimportant or that it is trivial to add them to GEDANKEN without destroying the generality of the language; this is a major theoretical problem.

The originality of GEDANKEN lies primarily in the language features which have been excluded, and the main aim of this paper is to demonstrate that these exclusions (except typelessness) do not impair generality. For this purpose, we include extensive programming examples.

A formal definition of GEDANKEN is given in [7]. A complete but extremely inefficient implementation has been produced by translating this formal definition into LISP; this implementation has been used to check all examples given in this paper.

* Applied Mathematics Division. Work performed under the auspices of the US Atomic Energy Commission.

After describing the syntax of the language and the types of values which are manipulated during program execution, we discuss the applicative part of the language, i.e. the evaluation of expressions and the application of functions. Finally, the imperative aspects, such as references, assignment, labels, and jumps, will be introduced.

Syntax

Although the importance of GEDANKEN lies in its semantics, a definite syntax must be specified so that programming examples can be given. A GEDANKEN program is a sequence of *tokens* separated by zero or more blanks, with at least one blank used as a separator whenever the juxtaposition would otherwise be ambiguous. The tokens are sequences of characters classified as follows:

constants digit strings (denoting integers), quoted strings
reserved words AND, OR, IF, THEN, ELSE, CASE, OF, IS, ISR
identifiers all other alphanumeric strings beginning with a letter
punctuation tokens λ , = : ( ) ; :=

Certain *predefined* identifiers have standard meanings. These include: TRUE, FALSE, LL, and UL, which denote specific primitive values; ERROR, which denotes a built-in label value causing program termination; and the names of all built-in functions. (These predefined identifiers differ from reserved words in that the programmer can override the standard meanings by declarations.)

The set of token sequences which are well-formed GEDANKEN programs is specified by the context-free grammar (over an infinite vocabulary of tokens) in Table I. The syntactic variables in this grammar are subscripted to distinguish among phrases with a similar semantic role but different levels of precedence. Thus phrases of the classes ⟨exp$_0$⟩, · · · , ⟨exp$_6$⟩ are all called *expressions*, while phrases of the classes ⟨pform$_0$⟩ and ⟨pform$_1$⟩ are called *parameter forms*. The notation $\{\alpha\}^*$ is used to indicate an arbitrary number (including zero) of occurrences of the string α.

It should be noted that a block can consist of a single expression; this permits any expression to be parenthesized without changing its semantics.

TABLE I. A GRAMMAR FOR GEDANKEN

⟨exp$_0$⟩ ::= ⟨constant⟩ | ⟨identifier⟩ | (⟨block⟩)
⟨exp$_1$⟩ ::= ⟨exp$_0$⟩ | ⟨function designator⟩
⟨function designator⟩ ::= ⟨exp$_0$⟩ ⟨exp$_1$⟩
⟨exp$_2$⟩ ::= ⟨exp$_1$⟩ | ⟨exp$_1$⟩ = ⟨exp$_2$⟩
⟨exp$_3$⟩ ::= ⟨exp$_2$⟩ | ⟨exp$_2$⟩ AND ⟨exp$_3$⟩
⟨exp$_4$⟩ ::= ⟨exp$_3$⟩ | ⟨exp$_3$⟩ OR ⟨exp$_4$⟩
⟨exp$_5$⟩ ::= ⟨exp$_4$⟩ | ⟨conditional exp⟩ | ⟨lambda exp⟩ | ⟨exp$_4$⟩ := ⟨exp$_5$⟩
⟨conditional exp⟩ ::= IF ⟨exp$_6$⟩ THEN ⟨exp$_6$⟩ ELSE ⟨exp$_5$⟩
⟨lambda exp⟩ ::= λ ⟨pform$_0$⟩ ⟨exp$_5$⟩
⟨exp$_6$⟩ ::= ⟨exp$_5$⟩ | ⟨sequence exp⟩ | ⟨case exp⟩
⟨sequence exp⟩ ::= ⟨empty⟩ | ⟨exp$_5$⟩, ⟨exp$_5$⟩ {, ⟨exp$_5$⟩}*
⟨case exp⟩ ::= CASE ⟨exp$_6$⟩ OF ⟨exp$_5$⟩ {, ⟨exp$_5$⟩}*
⟨pform$_0$⟩ ::= ⟨identifier⟩ | (⟨pform$_1$⟩)
⟨pform$_1$⟩ ::= ⟨pform$_0$⟩ | ⟨sequence pform⟩
⟨sequence pform⟩ ::= ⟨empty⟩ | ⟨pform$_0$⟩, ⟨pform$_0$⟩ {, ⟨pform$_0$⟩}*
⟨decl⟩ ::= ⟨pform$_1$⟩ IS ⟨exp$_6$⟩
⟨recursive decl⟩ ::= ⟨identifier⟩ ISR ⟨lambda exp⟩
⟨label⟩ ::= ⟨identifier⟩ :
⟨statement⟩ ::= {⟨label⟩}* ⟨exp$_6$⟩
⟨block⟩ ::= {⟨decl⟩;}* {⟨recursive decl⟩;}* {⟨statement⟩;}* ⟨statement⟩
⟨program⟩ ::= ⟨block⟩

Primitive Values and Functions

The items of data which are manipulated during the execution of a GEDANKEN program are called *values*. The set of all values is partitioned into seven *types*: *integers*, *Booleans*, *characters*, and *atoms* (*collectively called primitive values*), and *functions*, *references*, and *label values* (collectively called *nonprimitive values*). (Floating-point numbers are excluded, but their inclusion would not raise any significant problems.) Although the language does not contain type declarations, a complete set of built-in functions is available for testing the type of a value during program execution.

Among the primitive values, only *atoms* are unusual; they are similar to atoms in LISP, except that they lack property lists and print names. More precisely, the *atoms* are a denumerably infinite set of values which may be tested for equality, but which do not possess any ordering or arithmetic operations. Two particular atoms, denoted by the predefined identifiers LL and UL, play a special role in the language. Additional atoms are created by the built-in function ATOM, which returns a distinct atom each time it is applied.

A *function* is a value which may be *applied* to another value called its *argument*. When so applied, the function will either: (i) return a value called its *result*, (ii) transfer control to a label value without returning a result, (iii) cause an error stop, or (iv) initiate a nonterminating computation. (The application of a function may also alter the state of a computation by producing various *side effects*, which will be discussed later.) The set of arguments for which a function will return a result is called the *domain* of the function. A number of *built-in* functions are provided which may be used without being defined; additional "user-defined" functions are produced by the evaluation of various expressions.

(Proper procedures, in the sense of ALGOL, are not provided in GEDANKEN, since they are equivalent to functions which execute useful side effects but return an irrelevant result. Functions with multiple arguments are not provided, since they are equivalent to functions whose arguments are sequences, as described below.)

The functional approach to data structures is reflected in the absense of a distinct type of value corresponding to the conventional notion of a vector or array; the analogous values in GEDANKEN are functions. Thus we will use the word "vector" to denote those functions which are logically equivalent to conventional vectors.

It is evident that the domain of a GEDANKEN function which is a vector must include a finite set of consecutive integers; these integers are the analogue of the subscripts of a conventional vector. But a conventional vector also has the property that its set of subscripts is explicit; i.e. there must be some method of testing the vector to determine its least and greatest subscripts. To reflect this property in GEDANKEN, we require that the domain of a vector must include, in addition to the subscript set, the atoms LL and UL, and that the results of applying the vector to LL and UL must be the least and greatest subscripts.

This leads to the following definition. A function F is called a *vector* whenever: (1) its domain includes the atoms LL and UL; (2) the results of applying F to LL and UL are integers such that $F(UL) \geq F(LL) - 1$; (3) the domain of F includes all integers i such that $F(LL) \leq i \leq F(UL)$.

If F is a vector, then the integers F(LL), F(UL), and $F(UL) - F(LL) + 1$ are called the *lower limit*, *upper limit*, and *length* of F, respectively, and for each integer i such that $F(LL) \leq i \leq F(UL)$, the result of applying F to i is called the ith *component* of F.

A vector is called a *sequence* if its lower limit is 1.

Although a vector is a kind of function, and a sequence is a kind of vector, neither "vector" nor "sequence" is a "type" in the usual sense, since one cannot write a program which will test whether an arbitrary function is a vector or a sequence. Certain operations in GEDANKEN (e.g. evaluation of sequence expressions or application of the built-in function VECTOR) are guaranteed to produce vectors, but equally valid vectors may also be produced by more general mechanisms (e.g. evaluation of lambda expressions). Vectors produced in the latter manner are said to be *implicit*.

(The realization of vectors in GEDANKEN is in contrast to several languages, such as PAL, in which subscript limits are obtained by applying built-in functions to vectors. In the latter approach vectors are not purely functional, since they are amenable to other operations than application. The practical effect is to prohibit implicit vectors.)

The existence of sequences in GEDANKEN justifies the elimination of functions with multiple arguments. The analogue of a conventional function with k arguments, when either $k = 0$ or $k \geq 2$, is a function whose single argument is a sequence of length k. For example, the domain of the built-in function ADD is the set of sequences of length two whose components are both integers. (This approach is a direct borrowing from PAL.)

The remaining types of nonprimitive values, *references* and *label values*, will be defined later.

Applicative Semantics

To describe the semantics of GEDANKEN, we follow Landin [2] and Evans [3] in dividing the language into an applicative part, involving the evaluation of expressions and the application of functions, and an imperative part, involving assignment and control jumps. We first consider the applicative sublanguage, which is obtained by disregarding references, label values, and the operations which manipulate them.

Within this sublanguage, the basic operation is the *evaluation* of expressions. Since the evaluation of an expression will usually involve the evaluation of its subexpressions, the definition of this operation is inherently recursive. Also, when an expression contains free identifiers, its evaluation is only meaningful in the presence of some mapping of these identifiers into values; such a mapping is called an *environment* and is said to *bind* each identifier to a value.

A complete program is always evaluated in an environment which binds the predefined identifiers into their standard values. Whenever the evaluation of an expression e involves the evaluation of an immediate subexpression e', then, *unless* e is a lambda expression or a block, e' is evaluated in the same environment as e. The evaluation of lambda expressions and blocks (described in detail below) involves the concept of *extension*: if i is an identifier, v is a value, and η and η' are environments such that η' binds i to v and specifies the same binding as η for all other identifiers, then η' is called the *extension* of η formed by binding i to v.

We now describe the evaluation of each nontrivial form of expression. The application of a function to an argument is performed by a *function designator*:

$$\langle \text{function designator} \rangle ::= \underbrace{\langle \text{exp}_0 \rangle}_{\substack{\text{function} \\ \text{part}}} \quad \underbrace{\langle \text{exp}_1 \rangle}_{\substack{\text{argument} \\ \text{part}}}$$

which is evaluated by first evaluating its function part and its argument part to obtain values v_f (which must be a function) and v_a, and then applying v_f to v_a. (Since the argument part is evaluated before the function is applied, this form of evaluation is similar to call by value in ALGOL, rather than call by name.) Since function designators have a right-associative syntax, the usual composition of functions may be written without parentheses; e.g. F(G(X)) may be written as F G X.

Functions may be produced by the evaluation of *lambda expressions*:

$$\langle \text{lambda exp} \rangle ::= \lambda \ \langle \text{pform}_0 \rangle \ \underbrace{\langle \text{exp}_6 \rangle}_{\text{body}}$$

Basically, the value of a lambda expression is a function which (when it is applied to an argument at some later point during the computation) computes its result by binding the parameter form to its argument and then evaluating the body. More precisely, if f is the function obtained by evaluating $\lambda(p)e$ in the environment η, then the result of applying f to an argument a will be obtained by evaluating e in an environment which is the extension of η formed by binding p to a. (The meaning of binding p to a, when p is not an identifier, will be defined below.)

This binding mechanism is quite conventional (it is called FUNARG binding in LISP and is similar to the mechanism used in ALGOL and in PL/I), but a clear under-

standing of its implications is vital. There are two separate actions: (i) the evaluation of the lambda expression to produce a function, and (ii) the application of this function to its arguments. The body of the lambda expression is not evaluated until (ii), but the environment in which the body is evaluated is an extension of the environment used during (i) rather than (ii). As a result, when a lambda expression contains free identifiers, its evaluation in different environments will produce different functions. For example, in an environment where Y is bound to an integer k, the evaluation of λ(X) ADD(X, Y) produces a function which increases its argument by k.

Functions which are sequences may also be produced by the evaluation of *sequence expressions*:

⟨sequence exp⟩ ::= ⟨empty⟩ | ⟨exp$_6$⟩, ⟨exp$_6$⟩ {, ⟨exp$_6$⟩}*

Let n be the number of subexpressions. Then the sequence expression is evaluated by first evaluating its subexpressions to obtain values $v_1, \cdots, v_n$ and then producing a sequence of length n whose ith component (for $1 \leq i \leq n$) is v_i.

Because of their low precedence, sequence expressions are usually parenthesized, but the parentheses themselves do not indicate a sequence expression. Thus the expressions ( ) and (X, Y) both produce sequences, but (X) has the same value as X. There is no sequence expression which produces a sequence of length one, but such sequences can be produced by the built-in function UNITSEQ, which returns a sequence whose only component is the value of its argument.

As noted earlier, a function of n arguments ($n \neq 1$) is treated in GEDANKEN as a function of a sequence of length n. This suggests that when a function produced by a lambda expression expects to receive a sequence as its argument, the parameter form within the lambda expression should be able to bind several different identifiers to the components of the sequence. To provide this capability we extend the notion of a parameter form to include a *sequence parameter form* (which is a rough analogue of a formal parameter list in ALGOL):

⟨sequence pform⟩ ::= ⟨empty⟩ | ⟨pform$_0$⟩, ⟨pform$_0$⟩ {, ⟨pform$_0$⟩}*

The relevant semantics are given by defining (recursively) the *extension* of an environment η formed by binding an arbitrary parameter form p to a value v. This extension is computed as follows:

(1) If p is an identifier, then η is extended by binding p to v.
(2) If p has the form (p'), then η is extended by binding p' to v.
(3) If p is a sequence parameter form, $p_1, \ldots, p_n$ ($n \neq 1$), then v, which must be a function, is applied to each integer from 1 to n, and η is repeatedly extended by binding each p_i to the result of $v(i)$.

The syntax of sequence expressions and sequence parameter forms preserves conventional notation for functions of several arguments. Thus in the evaluation of (λ(X, Y) *body*) (3, 4), X is bound to 3 and Y is bound to 4. However, the sequence argument approach also provides useful unconventional capabilities, e.g. (λ(X, Y) *body*) (IF P THEN (3, 4) ELSE (5, 6)). More importantly, the ability to bind a single identifier to an entire sequence provides the equivalent of a function with an indefinite number of arguments, e.g. (λX *body*) (IF P THEN (3, 4) ELSE (5, 6, 7)).

GEDANKEN is similar to EULER [8] in treating all types of unlabeled statements as expressions. In particular, a *block* is a form of expression with a meaningful value:

⟨block⟩ ::= {⟨decl⟩;}* {⟨recursive decl⟩;}* {⟨statement⟩;}* ⟨statement⟩

where

⟨decl⟩ ::= ⟨pform$_1$⟩ IS ⟨exp$_6$⟩
⟨recursive decl⟩ ::= ⟨identifier⟩ ISR ⟨lambda exp⟩

Basically, a block is evaluated by first carrying out the bindings indicated by its declarations, recursive declarations, and labels, and then evaluating the statements in order from left to right. The value of the block is the value of the rightmost statement. The values of preceding statements are ignored; in the absence of imperative features, these statements have no effect.

More precisely, a block is evaluated as follows (we include the binding of labels although it is an imperative aspect of the language):

(1) For each declaration (⟨decl⟩), in order from left to right: the right side of the declaration is evaluated, and then the current environment is extended by binding the left side of the declaration to the value of the right side.

(2) The current environment is further extended by binding each identifier which occurs on the left of a recursive declaration (⟨recursive decl⟩), or as the label of a statement, to a distinct "dummy" value.

(3) The right side of each recursive declaration is evaluated, and its value replaces the corresponding dummy value.

(4) For each label, an apropriate label value is created and replaces the corresponding dummy value.

(5) The statements are evaluated in order from left to right.

(6) The value of the block is the value of the rightmost statement.

In steps 2 to 4, the device of binding identifiers to dummy values and then replacing the dummy values allows an environment to be cyclic, i.e. to bind an identifier to a value which is produced by evaluating a lambda expression (or label) in the same environment.

The essential difference between (nonrecursive) declarations and recursive declarations is that the right side of a declaration "feels" only the bindings caused by preceding declarations, while the right side of a recursive declaration feels the bindings caused by all declarations in the block, including implicit label declarations. Recursive declarations are needed to define recursive functions conveniently, including families of functions which call one another.

(They also permit the definition of functions which jump into the immediately enclosing block.)

Nonrecursive declarations are less essential, but they permit convenient constructions such as X IS ADD(X, 1). More important, their existence allows the right sides of recursive declarations to be limited to lambda expressions, so that meaningless constructions such as X ISR ADD(X, 1) are syntactically illegal.

Conditional expressions have the same meaning as in ALGOL. *Case expressions* have a rather unorthodox meaning (which is convenient for defining implicit sequences): CASE e_0 OF $e_1, \ldots, e_n$ is evaluated by first evaluating e_0 to obtain a value i; then if i is an integer satisfying $1 \leq i \leq n$, the value of the case expression is obtained by evaluating e_i; if i is LL or UL the value is 1 or n respectively; all other values of i give an error stop.

The remaining forms of expressions are most easily defined as abbreviations. Except for coercion (discussed later), they can be eliminated from a program by applying the following transformations:

$e_1 = e_2 \Rightarrow$ EQUAL(e_1 , e_2)
e_1 AND $e_2 \Rightarrow$ (IF e_1 THEN e_2 ELSE FALSE)
e_1 OR $e_2 \Rightarrow$ (IF e_1 THEN TRUE ELSE e_2)
$e_1 := e_2 \Rightarrow$ SET(e_1 , e_2)

The built-in function SET will be defined later. EQUAL tests the equality of primitive data, but if either component of its argument is a function or a label value, it will return FALSE. Its action on references will be described later.

Theoretically, nonrecursive declarations, sequence parameter forms, and sequence expressions can also be regarded as abbreviations. Their occurrences in a program can be eliminated by repeated application of the following equivalences:

p IS e; $b \Rightarrow (\lambda(p)(b))(e)$
$\lambda(p_1, \cdots, p_n)\ b$ (when $n \neq 1$)
$\quad \Rightarrow \lambda i(p_1$ IS i 1; $\cdots$; p_n IS i n'; $b)$
$e_1, \cdots, e_n$ (when $n \neq 1$)
$\quad \Rightarrow (i_1$ IS e_1 ; $\cdots$; i_n IS e_n ; λi(CASE i OF $i_1, \cdots, i_n$))

where n' is an integer constant whose value is n, and $i, i_1, \ldots, i_n$ are distinct identifiers which do not occur in the program being transformed.

It should be noted that GEDANKEN does not include certain features, such as infix arithmetic operators or **for** statements, which would enhance the conciseness of the language without expanding the range of programs which could be expressed. Such features could be added easily, but they are not germane to the basic purposes of the language.

Functional Data Structures

Even the applicative part of GEDANKEN is sufficient to demonstrate the power and flexibility which can be obtained by treating data structures functionally.

As a first example, consider LISP-like list structures. To define analogues of the LISP functions CONS, CAR, and CDR, we treat the two-field list cell produced by CONS as a function whose domain contains two elements (e.g. 1 and 2) and which maps these elements into the values of its CAR and CDR fields. This viewpoint leads directly to the definitions:

```
CONS IS λ(X, Y) λZ IF Z = 1 THEN X ELSE Y;
CAR IS λX X 1;
CDR IS λX X 2;
```

These definitions imply an ability to do list processing without special built-in functions. In a conventional list-processing system (e.g. compiled LISP 1.5 [1a and 1b] or some extensions of ALGOL [4, 9]) user-defined functions are restricted so that storage for the values of their identifiers obeys a stack discipline. Then list structures, which do not obey a stack discipline, must be allocated in a separate storage area, and built-in functions or operations must be provided for accessing this area. But in GEDANKEN, the user may develop list-processing by defining function-returning functions (such as CONS above) which violate a stack discipline. In effect, all storage is potentially list-structured.

Although the above approach is workable and theoretically attractive, it is more convenient to use sequence expressions to create list elements and direct application to obtain their subfields. Thus, we write (X, Y) instead of CONS (X, Y), X 1 instead of CAR X, and X 2 instead of CDR X. Following this approach, we introduce lists by first creating an atom to denote the empty list:

```
NIL IS ATOM( );
```

and then defining a list to be either the atom NIL or a sequence of length two whose second component is a list. The following functions will return the length of a list, find the ith element of a list, and append one list to another:

```
LISTLENGTH ISR λL IF L = NIL THEN 0
  ELSE INC LISTLENGTH L 2;
LISTELEM ISR λ(I, L) IF L = NIL THEN GOTO ERROR
  ELSE IF I = 1 THEN L 1 ELSE LISTELEM(DEC I, L 2);
APPEND ISR λ(X, Y) IF X = NIL THEN Y
  ELSE (X 1, APPEND(X 2, Y));
```

Hence INC and DEC are built-in functions which increase or decrease an integer by one.

As a second example, consider one-dimensional arrays. We have defined a type of function called a vector which is the analogue of a one-dimensional array, and we have introduced sequence expressions for creating vectors. But a sequence expression can only produce a vector which is a sequence, and it is inconvenient for producing very long vectors. What is needed is a function which will produce a vector from a functional specification of its components, i.e. which will accept another function, tabulate its results over a finite range, and return a "lookup" function for the resulting table.

Thus we define a function VECTOR which accepts an argument (L, U, F), where L and U are integers and F is a

function. If U < L, VECTOR returns an empty vector V such that V(LL) = L and V(UL) = L − 1. Otherwise, VECTOR evaluates F(I) for each integer I between L and U inclusive, and returns a vector V such that V(LL) = L, V(UL) = U and for L ≤ I ≤ U, V(I) is the value of F(I). The basic approach is to recur on the length of the vector, tabulating a single value (bound to T) at each level of recursion.

```
VECTOR ISR λ(L, U, F)
IF GREATER(L, U) THEN
  λ I IF I = LL THEN L ELSE IF I = UL THEN DEC L
    ELSE GOTO ERROR
ELSE (V IS VECTOR(L, DEC U, F); T IS F U;
  λ I IF I = UL THEN U ELSE IF I = U THEN T ELSE V I);
```

It is evident that this function, although theoretically correct, will be extremely inefficient in any reasonable implementation. For this reason, a built-in function VECTOR is provided which is defined to be equivalent to the function above (except for coercion).

(This question of efficiency may be clarified by considering implementation mechanisms. In a simple implementation, functions would possess two distinct internal representations: If a function was produced by evaluating a lambda expression, it would be represented by a "lambda record" containing a pointer to code which was compiled from the lambda expression plus values for each free identifier in the lambda expression (i.e. a representation of the environment in which the lambda expression was evaluated). On the other hand, if a function was created by evaluating a sequence expression or by the application of VECTOR, it would be represented by a "vector record" containing domain limit and indexing information plus a contiguous array of component values. It is evident that the above definition of VECTOR would yield a vector whose internal representation was a linked list of lambda records, each containing one component value, rather than a contiguous array.)

Using lists and vectors, we may illustrate our assertion that any process which accepts some data structure will accept any logically equivalent structure. Suppose that P is a function which expects a sequence as its argument, and

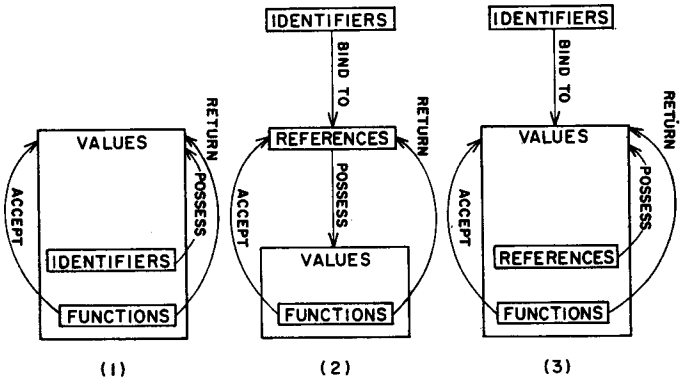

FIG. 1. Three approaches to assignment

that we wish to give it a sequence whose ith component is the ith element of a list L. This can be done in a conventional manner by evaluating P VECTOR (1, LISTLENGTH L, λ I LISTELEM(I, L)), which copies the elements of L into a contiguous array. But it is also possible to evaluate P MAKESEQFROMLIST L, where

```
MAKESEQFROMLIST IS λ L
λ I IF I = LL THEN 1 ELSE IF I = UL THEN LISTLENGTH L
  ELSE LISTELEM(I, L);
```

MAKESEQFROMLIST does not copy the components of L; instead, it returns an implicit sequence which will look up the appropriate element of L each time one of its components is accessed.

It is equally possible to produce an implicit list from a sequence:

```
MAKELISTFROMSEQ ISR λ S MLFS1(1, S);
MLFS1 ISR λ(I, S) IF GREATER (I, S UL) THEN NIL
  ELSE λ K (CASE K OF S I, MLFS1(INC I, S));
```

(Here MLFS1 is a subsidiary function which produces an implicit list from the subsequence of S that begins with the Ith component.)

The data structures shown so far have the limitation that once a structure has been created, its components or elements cannot be altered. To overcome this limitation we must introduce the imperative aspects of GEDANKEN.

References

In any programming language which permits assignment, there is a class of objects which are affected by assignment. We will call these objects *references*; other terms used commonly in the literature are "name" and "L-value." At any time during the execution of a program, each reference *possesses* some value. The effect of an assignment operation $r := v$ is to cause the reference denoted by r to possess the value denoted by v.

Within this definitional framework, there are at least three distinct approaches to assignment (see Figure 1):

(1) Identifiers are used as references. This approach is used in SNOBOL [10], where a form of indirect addressing is achieved by allowing identifiers to occur as values. Unfortunately, the approach does not mesh well with block structure; a discussion of the difficulties is given by Kain [11].

(2) References are distinct from either identifiers or values, and are interposed between all other value-denoting entities and their values. Thus the bindings of identifiers, the arguments and results of functions, and the components of vectors are all references, and the values denoted by these entities are actually the values possessed by the references. This approach is used in PAL, and to a large extent in FORTRAN and PL/I, except that in the latter languages function results are values, and identifiers may be bound directly to functions and label values, but not to primitive values. The approach meshes well with block structure but is rather inflexible; one moves from the applicative situation, where assignment is impossible, to

the opposite extreme, where every value-denoting entity can be affected by assignment.

(3) References are treated as a distinct type of value, so that any value-denoting entity can denote either a conventional value or a reference which in turn possesses a value. This approach is used in ALGOL 68 and BASEL. (BASEL also permits a form of assignment which alters identifier binding.) It is compatible with block structure and is more flexible than the previous approach, since the programmer can introduce references in just those contexts where he intends to do assignment. Advantages should accrue in both the optimization of data representations and the checking of erroneous assignment statements.

(The above categorization must be qualified by the fact that FORTRAN, PL/I, ALGOL 68, and BASEL all have type-declaration mechanisms which affect their treatment of assignment. A discussion of this interaction is beyond the scope of this paper.)

In GEDANKEN we have chosen to use the third approach to assignment. Thus we introduce a new, denumerably infinite set of values called *references*, and stipulate that each reference *possesses* some other value (which may itself be a reference). Three built-in functions are provided to manipulate references: REF, SET, and VAL. REF X returns a distinct reference each time it is applied; this reference is initialized to possess the value X. SET(R, X) (which can be abbreviated R := X) causes R (which must be a reference) to possess the value X, and also returns X; its action on R is an example of a *side effect*. VAL R returns the value possessed by R (which must be a reference).

For example, under the scope of the declaration X IS 3, the identifier X is bound to the integer 3, and this binding cannot be altered by assignment. Evaluation of the expression X := 4 would give an error, since 3 is not a reference. Analogously, under the scope of the declaration X IS REF 3, the identifier X is bound to the reference created by REF, and this binding cannot be changed by assignment. But now evaluation of X := 4 is legitimate, and causes the value possessed by the reference bound to X to change from 3 to 4. Thus in the execution of the block

```
(X IS REF 3; VAL X = 3; X := 4; VAL X = 4)
```

both equality predicates will be true.

The major difficulty with this approach is the frequent necessity for using the function VAL. For example, under the scope of the declarations X IS REF 3; Y IS REF 4; one would write ADD(VAL X, VAL Y) rather than ADD(X, Y), since ADD acts upon integers rather than references. To alleviate this difficulty, we introduce *coercion* conventions into GEDANKEN; i.e. we stipulate that references will be replaced by their values in certain contexts which would otherwise be meaningless.

Specifically, let COERCE be the function

```
COERCE ISRλ X IF ISREF X THEN COERCE VAL X ELSE X;
```

(which is available as a built-in function), and define "to coerce X" to mean the replacement of X by COERCE X. Then:

(1) All built-in functions which would otherwise be meaningless coerce their argument or the appropriate components of their arguments. For example, ADD(X, Y) is equivalent to ADD(COERCE X, COERCE Y), but ISREF X is not equivalent to ISREF COERCE X, nor VAL X to VAL COERCE X.

(2) REF X coerces X, SET(R, X) (and therefore R := X) coerces X, and EQUAL(X, Y) (and therefore X = Y) coerces both X and Y. Since these functions would each be meaningful for references without coercion, analogous noncoercing functions, named NCREF, NCSET, and NCEQUAL, are also provided. NCREF and NCSET permit references to possess values which are also references. NCEQUAL can be used to determine whether two values are the same reference.

(3) Conditional and case expressions coerce the values of their leftmost subexpressions.

(4) Expressions involving AND and OR coerce the values of both their subexpressions.

(5) A function designator coerces the value of its function part.

(6) When a sequence parameter form $p_1, \ldots, p_n$ is bound to a value a, each p_i will be bound to (COERCE a) (i).

(7) Vectors which are created by evaluating sequence expressions or by application of the built-in functions VECTOR or UNITSEQ will coerce their argument.

Despite their ad hoc appearance, most of these coercion rules are instances of the general principle that coercion should only occur in situations which would otherwise give an error termination. The exceptions are rules (2) and (4), which are simply concessions to conventional notation.

Data Structures with Embedded References

The utility of references becomes apparent when reference-returning functions are used to embed references within data structures, yielding structures which can be altered by assignment.

This approach provides precise control over the ways in which data structures can be altered. Thus the GEDANKEN equivalent of an ALGOL-like one-dimensional array is a vector whose components are references, e.g.

```
X IS VECTOR(1, 100, λ I REF 0);
```

Under the scope of this declaration, assignment can be made to the components of X, e.g. X(7) := 10, but not to X itself. In particular, the subscript limits X LL and X UL are fixed by the declaration.

On the other hand, the equivalent of a string variable is provided by a reference whose value is a vector:

```
S IS REF VECTOR(1, 100, F);
```

Here assignment can be made to S itself (possibly changing the subscript limits) but not to its components.

A second consequence of the reference concept is the

ability to define data structures or sets of data structures which share elements, in the sense that assignment to one element will affect another. Consider a square matrix M. We could define M as a vector of vectors, i.e.

```
M IS VECTOR(1, 10, λ I VECTOR(1, 10, λ J REF 0));
```

but this leads to the inconvenience of referring to an element of M by (M I) J. It is more natural to define M as a reference-returning function of pairs of integers:

```
M IS (M1 IS VECTOR(1, 10, λ I VECTOR(1, 10, λ J REF 0));
  λ(I, J) (M1 I) J);
```

so that an element is referred to as M(I, J). Now consider the additional declarations:

```
MT IS λ(I, J) M(J, I); MD IS λ I M(I, I);
```

Here MT and MD denote the transpose and diagonal of M, in the sense that assignment to an element of one matrix affects the corresponding elements of the others.

Elements may also be shared within the same data structure. For example,

```
S IS(S1 IS VECTOR(1, 10, λ I VECTOR(1, I, λ J REF 0));
  λ(I, J) IF NOT GREATER(J, I) THEN (S1 I) J ELSE (S1 J) I);
```

defines a symmetric matrix in which assignment to S(I, J) also alters S(J, I).

The embedding of references in list structures also provides control over the ways in which these structures may be altered. An example is the property list, which is a list of property-value pairs subject to two operations: the value paired with a given property may be looked up; or the value paired with a given property may be changed, adding a new pair to the list if the property is not already present. It is evident that references must occur in the property list at two points: each value must be a reference, so that it can be changed; and the entire list must be a reference, so that new pairs can be added.

The following function manipulates such property lists. Given a property P and a (reference to a) property list L, PROPVAL(P, L) searches L for an occurrence of P. If P is found, the reference paired with P is returned. Otherwise, a pair consisting of P and a new reference (initialized to zero) is added to L, and the new reference is returned. The argument P is coerced.

```
PROPVAL IS λ(P, L)
  (P IS COERCE P;
  SEARCHL ISR λ X
    IF X = NIL THEN
      (NEWV IS REF 0; L := ((P, NEWV), VAL L); NEWV)
    ELSE IF (X 1) 1 = P THEN (X 1) 2 ELSE SEARCHL X 2;
  SEARCHL VAL L);
```

An application of this function can occur on either side of an assignment operation; on the right side it will act to look up a value, on the left side it will act to alter a value.

A further step can be taken by viewing the property list itself as a reference-returning function which accepts a property and returns a reference to the corresponding value. The following function (of no arguments) returns such *functional* property lists:

```
MAKEPROPLIST IS λ( ) (L IS REF NIL; λ P PROPVAL(P, L));
```

Each application of MAKEPROPLIST returns a new instance of PROPVAL, with L bound to a private "own variable." Since a property can be any primitive value, a functional property list is similar to a reference-valued vector, except that it has an indefinite domain. Indeed, functional property lists can be used to provide an efficient implementation of sparse vectors.

As a final example of the use of references, suppose that READ is a function such that each application of READ produces the next item of data from some input stream, and that we wish to produce an implicit list of the successive items in the stream. The following function (of no arguments) returns such a list:

```
MAKERLIST ISR λ( )
  (B IS REF 0; λ I
    (IF B = 0 THEN B := (READ( ), MAKERLIST( )) ELSE ( );
      B I));
```

The result of MAKERLIST is an implicit list (whose implicit length is infinite) which only applies READ as items of data are actually needed, and only stores previously read items which are still accessible.

Implicit References

The utility of implicit data structures suggests the introduction of an analogous facility for references. Thus we introduce the concept of an *implicit reference*, i.e. a value whose external appearance is the same as a reference, but which may carry out an arbitrary computation each time it is set or evaluated. (Implicit references are related to doublets in POP-2 [12].)

To specify an implicit reference, the programmer must provide two functions: a "setting function" S which will be executed each time a value is assigned to the implicit reference, and an "evaluating function" V which will be executed each time the implicit reference is evaluated. Thus an implicit reference is produced by applying the built-in function IMPREF(S, V), where S and V may be arbitrary functions of one and zero arguments respectively. Each application of IMPREF produces a distinct implicit reference, and these implicit references satisfy the predicate ISREF and are coerced in the same manner as conventional references. But the effect of SET or VAL on an implicit reference is to execute S or V. Specifically, if R is the result of IMPREF(S, V), then

```
NCSET(R, X) = (S X; X)
SET(R, X) = (X IS COERCE X; S X; X)
VAL R = V ( )
```

To illustrate the use of implicit references, consider the problem of protecting a reference-valued vector. Suppose that P is a function which accepts a vector whose components are references. We wish to apply P to such a vector V, but to protect the components of V from being

affected by P; i.e. we want these components to revert to their original values after the application of P is finished. The simplest approach is to copy V by executing P VECTOR(V LL, V UL, λ I REF V I), but this will be inefficient if V is large and only a few components are reset by P. An alternative approach is to maintain a "change list" of the components of V which have been altered by P. This may be done by executing P PSEUDOCOPY V, where

```
PSEUDOCOPY IS λ V
  (CL IS REF NIL;
  SEARCHCL ISR λ(X, I, F, G) IF X = NIL THEN G( )
    ELSE IF (X 1) 1 = I THEN F (X 1) 2
    ELSE SEARCHCL(X 2, I, F, G);
  λ I (I IS COERCE I;
    IF I = LL THEN V LL ELSE IF I = UL THEN V UL
    ELSE IF NOT ISINTEGER I OR GREATER(V LL, I)
      OR GREATER(I, V UL)
      THEN GOTO ERROR
    ELSE IMPREF(
      λ X SEARCHCL(VAL CL, I, λ R NCSET(R, X),
        λ( ) (CL := ((I, NCREF X), VAL CL)),
      λ( ) SEARCHCL(VAL CL, I, VAL, λ( ) VAL V I))));
```

The result of PSEUDOCOPY is an implicit vector whose components are implicit references. Internally, CL is a reference to the change list, which is a list of pairs, each containing an integer argument of some altered component and a reference to the current value of that component. SEARCHCL is a subsidiary function which searches a change list X for a pair beginning with the integer I. If such a pair is found, SEARCHCL returns the result of F applied to the reference paired with I; otherwise SEARCHCL returns the result of G, which is a function of no arguments. (The noncoercing functions NCSET and NCREF are used to allow the values possessed by the components of V to be references.)

Label Values

The final type of value used in GEDANKEN is the *label value*. These values are created during execution of a block containing labeled statements, and are used as arguments to the built-in function GOTO, which never returns but instead causes a transfer of control to the computational state represented by the label value.

A more precise description requires introducing a model of the interpretation of GEDANKEN by an abstract machine. A complete description of such a model (given in [7]) is beyond the scope of this paper, but the following aspects are relevant to an understanding of the label and GOTO mechanisms:

During the execution of a program (at any instant when a statement is about to be evaluated) the *state* of the abstract interpreter will include the following entities:

(1) A *control*, which gives a list of the statements remaining to be evaluated in the current block.

(2) An *environment*, which gives the identifier bindings to be used in the current block.

(3) A *dump*, which specifies the computations to be performed after the current block is completed. The dump is a pushdown stack containing an entry for each block and lambda-expression body whose evaluation is incomplete; each entry contains a control and an environment (plus additional information which is needed to describe partially evaluated compound expressions).

(4) A *memory*, which specifies the mapping of references into their values.

A label value consists of a control, an environment, and a dump. During the evaluation of a block, immediately before the first statement is evaluated, a label value is created for each label in the block; each label value contains a list of the statements between the corresponding label and the block end, plus the current environment (including the bindings of the labels themselves) and dump.

When the built-in function GOTO is applied to a label value, the current control, environment, and dump are replaced by the constituents of the label value, and execution continues with the first statement of the new control. The memory is not altered.

This mechanism permits jumps within the same block (which leave the environment and dump unchanged) or to higher level blocks, with the same effect as in ALGOL. But the fact that label values can be possessed by references or returned by functions also provides the ability to jump back into a block after it has been exited from. It is this capability which allows the construction of coroutines.

Coroutines

A coroutine is a procedure which can relinquish control to its calling program and later be reactivated to continue computation. The simplest situation is that of two procedures, each of which treats the other as a subroutine.

As an example, suppose that COMPILE is a procedure which produces a succession of data items called instructions, outputting each instruction by applying a function OUT, and that ASSEMBLE is a procedure which accepts a succession of instructions, inputting each instruction by applying a function IN. If OUT and IN are arguments to COMPILE and ASSEMBLE respectively, we have

```
COMPILE ISR λ OUT ( ··· OUT X ··· );
ASSEMBLE ISR λ IN ( ··· X := IN( ) ··· );
```

We now want to couple these procedures so that ASSEMBLE receives the output of COMPILE. Specifically, we want to run ASSEMBLE until it requests input, than run COMPILE until it produces the required output, then run ASSEMBLE again, etc. The necessary program can be written by using label-valued references which are global to both IN and OUT:

```
(LC IS REF 0;  LA IS REF 0;  INST IS REF 0;
LC := LC1; ASSEMBLE(λ( ) (LA := LA1;  GOTO LC;
  LA1: VAL INST)); GOTO DONE;
LC1:  COMPILE(λ X (LC := LC2;  INST := X;  GOTO LA;
  LC2:)); GOTO ERROR;
DONE:);
```

Here LA and LC are label-valued references saving the current states of ASSEMBLE and COMPILE, and INST is a third reference used to hold the instruction being transmitted from COMPILE to ASSEMBLE. If COMPILE finishes while ASSEMBLE is still waiting for another instruction, an error stop occurs.

Nondeterministic Algorithms

Label values in GEDANKEN are closely related to "processes" in simulation languages such as SIMULA [13a and 13b]; both are mechanisms which allow the state of a suspended computation to be saved as an item of data. The essential difference is that further execution of a computation which was saved as a process causes the process to be updated, while further execution of a computation saved as a label value leaves the label value unchanged. Thus label values can be used to repeatedly initiate execution from the same state.

This capability can be used to program a mode of execution for nondeterministic algorithms [14] in which alternative paths are pursued concurrently. A simple example is nondeterministic parsing. It is fairly straightforward to convert a context-free grammar into a recursive parsing function. Unfortunately, for many grammars this function will contain nondeterministic branches, i.e. points at which a conditional branch must be performed although the current state of the parse is insufficient to determine this branch.

When such nondeterminism exists, parsing can be accomplished by simulating a finite set of independent parsers, all accepting the same input string and obeying the same program, but with different control states. When a parser encounters a nondeterministic branch, it expands into two separate parsers; when a parser reads an input character which is inconsistent with its control state, it is deleted.

Specifically, we assume that PARSE(IN, AMB, FAIL) is a function which accepts two functions IN and AMB, and a label value FAIL, and returns some representation of a successful parse. The function IN, of no arguments, is applied by PARSE to read each character of the input string. The function AMB, whose argument is a label value, is applied to execute a nondeterministic branch; one side of the branch returns from AMB while the other jumps to the label-valued argument. PARSE jumps to the label value FAIL when it encounters an inconsistent character. We assume that PARSE does not set any references, or at least that it does not expect the value of any reference to be preserved across an application of IN or AMB.

The following program carries out the concurrent execution of PARSE, synchronizing the independent parsers by their reading of characters:

```
(C IS REF NIL; W IS REF NIL; R IS REF NIL;
   CHAR IS REF NIL;
 C := (PARSE(λ( ) (W := (L1, VAL W); GOTO CONT;
       L1: VAL CHAR),
     λ L2 (R := (L2, VAL R)), CONT),
   VAL C);
CONT: IF R = NIL AND W = NIL THEN GOTO DONE
   ELSE IF R = NIL
     THEN (CHAR := READCHAR( ); R := W; W := NIL)
   ELSE ( );
 (L IS R 1; R := R 2; GOTO L);
DONE: VAL C)
```

Each independent parser is represented by a label value if it has not completed its parse, or by its result if it has completed its parse. The finite set of parsers is maintained by the values of the references C, W, and R. C gives a list of the results of completed parses, W gives a list of label values representing the parsers which are waiting for the next character, and R gives a similar list for the parsers which are ready for execution before reading the next character. The reference CHAR keeps track of the current character, and is updated by the built-in function READCHAR. The label CONT is reached whenever execution is to be switched from one parser to another. The final value of the block is the list of completed parses; the input string is ill formed, well formed, or ambiguous depending upon whether this list has zero, one, or more than one element.

(This approach to parsing is basically the same as that used in the COGENT programming system [15a and 15b]. It is presented here as an illustration of the generality of GEDANKEN, but it does not represent a significant advance in the field of parsing techniques. Although it is reasonably efficient for a large class of unambiguous grammars, at least if the function PARSE is carefully constructed, some ambiguous grammars will cause an exponential growth in the number of parsers and are better treated by other methods, such as that of Earley [16].)

Limitations and Possible Extensions

The goal of applying the basic principles of GEDANKEN to the design of an efficient general purpose programming language raises several interesting research problems:

(1) *Addition of Type Declarations.* The most natural approach is probably an extension of Hoare's concept of record classes [9]. The programmer would be able to declare an arbitrary number of disjoint function, reference, and label classes, and would specify the range of each identifier, function result, and reference value to be some union of such classes (and/or predefined classes of primitive values). All functions in the same class would have the same domain-range relation, and all references in the same class would have the same set of possible values.

However, the functional approach to data structures will require unusual flexibility in the specification of the domain-range relations of functions. If an inhomogeneous data structure such as a record is to be treated as a function, then it must be possible to specify that the range of such a function depends on its argument. For example, the set of lists of integers would be the union of the set {NIL} with a class of functions with domain (1, 2) which map 1 into an integer but map 2 into a list of integers.

An elaboration of this approach to type, limited to a purely applicative language, is described in [17].

(2) *Open Functions.* Efficient implementation of functional data structures will require that certain functions be compiled into open code, i.e. that function designators should be replaced by modified copies of the corresponding lambda-expression body, and that these copies should then be simplified to take advantage of constant arguments. This capability could be provided by a macro-definitional facility. A second approach, more in keeping with the spirit of GEDANKEN, would be to permit certain lambda expressions to be given an OPEN attribute.

This raises the question of whether a compiler could determine automatically when a designator of a lambda-defined function should be replaced by a copy of the function body. One might conjecture that such an expansion could be performed for any function which was defined by a nonrecursive declaration. Unfortunately, this conjecture is disproved by the existence of a nonrecursive *fixed-point function*:

```
Y IS λG (U IS λV G(λX (V V) X); U U);
```

which can be used to convert any simply recursive function (i.e. a function which calls itself directly but not indirectly via other functions) into an equivalent nonrecursive function [18].

Thus suppose a recursive function F is defined by F ISR *b*, where F is the only identifier which occurs free in *b*. Let F1 be the nonrecursive function defined by F1 IS λF (*b*). Then the function (Y F1) can be shown to be equivalent to F, with the same domain of termination. Moreover, the expansion of a function designator such as (Y F1) X by repeated substitution of the definitions of Y and F1 will never terminate.

(3) *Storage Allocation.* A serious drawback of the principle of completeness is the elimination of any run-time stack discipline, so that all data storage must be recovered by garbage collection. This problem might be alleviated by adding language facilities for indicating contexts where a stack discipline is applicable. Even without such facilities, it may be possible to determine by program analysis, particularly with appropriate type declarations, situations where storage can be recovered without garbage collection.

(4) *Side Effects.* In the applicative subset of GEDANKEN, the immediate subexpressions of a function designator or a sequence expression can be evaluated in any order, or the steps of their evaluation can be intermixed, without affecting the result or termination of any program. This property, which is obviously desirable for code optimization or multiprocessing, is destroyed by the introduction of assignment, since subexpressions can execute interfering side effects.

The situation is exacerbated by the introduction of label values, since then the order of evaluation can affect the number of times a subexpression is executed. The program

```
(X IS REF 0; (X := INC X, GOTO L); L:  VAL X)
```

produces one with left-to-right evaluation of the sequence expression, but produces zero with right-to-left evaluation. Label-valued references lead to more paradoxical programs, such as

```
(X IS REF 0; L IS REF 0; M IS REF 0; L  := L1;
  (X := INC X, (M := M1; M1:  GOTO L));
  L1:  L := L2; GOTO M; L2:  VAL X)
```

which produces one with left-to-right evaluation, zero with right-to-left evaluation, and possibly two with intermixed evaluation.

This problem is common to a wide variety of languages. One either imposes a fixed order of evaulation, as in ALGOL 60 or GEDANKEN, or permits a significant class of well-formed programs to have indeterminate interpretations, as in ALGOL 68 or PL/I. But a more flexible approach might be possible, e.g. a limited form of imperative features which could be added to an applicative language without destroying order-of-evaluation independence.

(5) *Other Label-Value Problems.* Label-valued references can easily cause the preservation of data which will no longer be accessed by a computation. If L is a label-valued reference, then GOTO L will cause execution to proceed from the computational state denoted by L. But the unchanged state must also be saved in case GOTO L is executed again before the value of L is changed. If, in fact, such a repeated jump cannot occur, then information will be saved unnecessarily unless the programmer goes to the trouble of resetting L immediately after the original jump. (As an example, the program for linking the coroutines COMPILE and ASSEMBLE will preserve the states of these routines unnecessarily.)

Presumably, it would be better to force the programmer to extra trouble in order to preserve, rather than discard, a reactivated computational state. This might be accomplished by adapting the concept of "process" used in simulation languages, and providing a basic function for copying processes. However, it is not clear how to combine the process concept with an ALGOL-like use of label values in a clean manner which does not violate the principle of completeness.

A further difficulty is the inability of a label value to preserve the values of references (i.e. the memory). In the nondeterministic parser described earlier, the restriction on the use of references in the function PARSE arises from this problem.

(6) *Secondary Storage and File Management.* Even with open functions and sophisticated code optimization, it may be intolerably inefficient to impose a purely functional approach on all data structures. But the functional approach still holds considerable promise for the treatment of large structures which require secondary storage. A stated, but usually unmet goal of most data management systems is the complete separation of the logical properties of a file from its physical representation. A natural approach to this goal would be to equate a logical file with a collection of functions for accessing the file, and to permit these functions to be implicit.

Acknowledgments. The author wishes to thank Dr. M. D. MacLaren of Argonne National Laboratory and Professor Arthur Evans, Jr., of Massachusetts Institute of Technology for their stimulating discussions and helpful suggestions.

RECEIVED APRIL, 1969; REVISED OCTOBER 1969; FEBRUARY, 1970

REFERENCES

1a. MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, Pt. I. *Comm. ACM 3*, 4 (Apr. 1960), 184–195.

1b. ——, ET AL. LISP 1.5 programmers manual. MIT Press, Cambridge, Mass., 1962.

2. LANDIN, P. J. The next 700 programming languages. *Comm. ACM 9*, 3 (Mar. 1966), 157–166.

3. EVANS, A. PAL—A language designed for teaching programming linguistics. Proc. ACM 23rd Nat. Conf. 1968, Brandin Systems Press, Princeton, N.J., pp. 395–403.

4. VAN WIJNGAARDEN, A. (Ed.), MAILLOUX, B. J., PECK, J. E. L., AND KOSTER, C. H. A. Report on the algorithmic language ALGOL 68. MR 101, Mathematisch Centrum, Amsterdam, Feb., 1969.

5. CHEATHAM, T. E., JR., FISCHER, A., AND JORRAND, P. On basis for ELF—An extensible language facility. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33 Pt. 2, MDI Publications, Wayne, Pa., pp. 937–948.

6. BALZER, R. M. Dataless programming. Proc. AFIPS 1967 Fall Joint Comput. Conf. Vol. 31, MDI Publications, Wayne, Pa., pp. 535–544.

7. REYNOLDS, J. C. GEDANKEN—A simple typeless language which permits functional data structures and coroutines. ANL-7621, Argonne Nat. Lab., Argonne, Ill., Sept. 1969.

8. WIRTH, N., AND WEBER, H. EULER—A generalization of ALGOL and its formal definition: Pt. I, Pt. II. *Comm. ACM 9*, 1 and 2 (Jan., Feb. 1966), 13-25, 89–99.

9. WIRTH, N., AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9*, 6 (June 1966), 413–432.

10. FARBER, D. J., GRISWOLD, R. E., AND POLONSKY, I. P. The SNOBOL3 programming language. *Bell Syst. Tech. J. 45* (July-Aug. 1966), 895–944.

11. KAIN, R. Y. Block structures, indirect addressing, and garbage collection. *Comm. ACM 12*, 7 (July 1969), 395–398.

12. BURSTALL, R. M., AND POPPLESTONE, R. J. POP-2 reference manual. In *Machine Intelligence 2*, E. Dale and D. Michie (Eds.), American Elsevier, New York, 1968, pp. 205–246.

13a. DAHL, O. J., AND NYGAARD K. SIMULA—An ALGOL-based simulation language. *Comm. ACM 9*, 9 (Sept. 1966), 671–678.

13b. ——, MYHRHAUG, B., AND NYGAARD, K. SIMULA 67 common base language. Publ. No. S-2, Norwegian Computing Center, Oslo, May 1968.

14. FLOYD, R. W. Nondeterministic algorithms. *J. ACM 14*, 3 (Oct. 1967), 636–644.

15a. REYNOLDS, J. C. An introduction to the COGENT programming system. Proc. ACM 20th Natl. Conf., 1965, pp. 422–436.

15b. ——, COGENT programming manual. ANL-7022, Argonne Nat. Lab., Argonne, Ill., Mar. 1965.

16. EARLEY, J. An efficient context-free parsing algorithm. *Com. ACM 13:* 2 (Feb. 1970), 94–102.

17. REYNOLDS, J. C. A set-theoretic approach to the concept of type. Working paper, NATO Conf. on Techniques in Software Engineering, Rome, Oct. 1969.

18a. EVANS, A. Private communication.

18b. MORRIS, J. H. Lambda-calculus models of programming languages. MAC-TR-57, Project MAC, MIT, Cambridge, Mass., Dec. 1968.